Forget to remember
Remember to forget

VLC = Vision Learning Control

Long Short Term Memories and Gated Recurrent Units

Jonathon Hare

Vision, Learning and Control
University of Southampton

Many of the images and animations used here were made by Adam Prügel-Bennett.

Recap: An RNN is just a recursive function invocation

- $\boldsymbol{y}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{c}(t-1)|\boldsymbol{W})$
- and the state $\boldsymbol{c}(t) = \boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{c}(t-1)|\boldsymbol{W})$
- If the output $\boldsymbol{y}(t)$ depends on the input $\boldsymbol{x}(t-2)$, then prediction will be

$$\boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{g}(\boldsymbol{x}(t-1), \boldsymbol{g}(\boldsymbol{x}(t-2)|\boldsymbol{W}), \boldsymbol{W}), \boldsymbol{W}), \boldsymbol{W})$$

- it should be clear that the gradients of this with respect to the weights can be found with the chain rule

Recap: An RNN is just a recursive function invocation

- $\boldsymbol{y}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{c}(t-1)|\boldsymbol{W})$
- and the state $\boldsymbol{c}(t) = \boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{c}(t-1)|\boldsymbol{W})$
- If the output $\boldsymbol{y}(t)$ depends on the input $\boldsymbol{x}(t-2)$, then prediction will be

$$\boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{g}(\boldsymbol{x}(t-1), \boldsymbol{g}(\boldsymbol{x}(t-2)|\boldsymbol{W}), \boldsymbol{W}), \boldsymbol{W}), \boldsymbol{W})$$

- it should be clear that the gradients of this with respect to the weights can be found with the chain rule
- The back-propagated error will involve applying $\boldsymbol{f}$ multiple times

Recap: An RNN is just a recursive function invocation

- $\boldsymbol{y}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{c}(t-1)|\boldsymbol{W})$
- and the state $\boldsymbol{c}(t) = \boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{c}(t-1)|\boldsymbol{W})$
- If the output $\boldsymbol{y}(t)$ depends on the input $\boldsymbol{x}(t-2)$, then prediction will be

$$\boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{g}(\boldsymbol{x}(t-1), \boldsymbol{g}(\boldsymbol{x}(t-2)|\boldsymbol{W}), \boldsymbol{W}), \boldsymbol{W}), \boldsymbol{W})$$

- it should be clear that the gradients of this with respect to the weights can be found with the chain rule
- The back-propagated error will involve applying $\boldsymbol{f}$ multiple times
- Each time the error will get multiplied by some factor a

Recap: An RNN is just a recursive function invocation

- $\boldsymbol{y}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{c}(t-1)|\boldsymbol{W})$
- and the state $\boldsymbol{c}(t) = \boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{c}(t-1)|\boldsymbol{W})$
- If the output $\boldsymbol{y}(t)$ depends on the input $\boldsymbol{x}(t-2)$, then prediction will be

$$\boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{g}(\boldsymbol{x}(t-1), \boldsymbol{g}(\boldsymbol{x}(t-2)|\boldsymbol{W}), \boldsymbol{W}), \boldsymbol{W}), \boldsymbol{W})$$

- it should be clear that the gradients of this with respect to the weights can be found with the chain rule
- The back-propagated error will involve applying $\boldsymbol{f}$ multiple times
- Each time the error will get multiplied by some factor a
- If $\boldsymbol{y}(t)$ depends on the input $\boldsymbol{x}(t-\tau)$ then the back-propagated signal will be proportional to $a^{\tau-1}$

Recap: An RNN is just a recursive function invocation

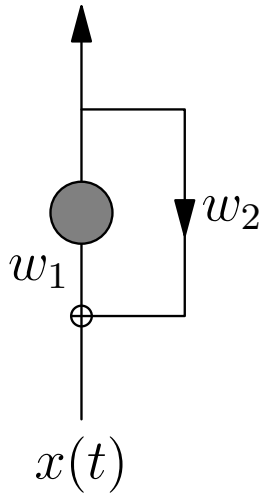
- $\boldsymbol{y}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{c}(t-1)|\boldsymbol{W})$
- and the state $\boldsymbol{c}(t) = \boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{c}(t-1)|\boldsymbol{W})$
- If the output $\boldsymbol{y}(t)$ depends on the input $\boldsymbol{x}(t-2)$, then prediction will be

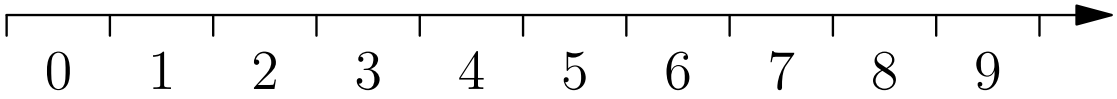
$$\boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{g}(\boldsymbol{x}(t-1), \boldsymbol{g}(\boldsymbol{x}(t-2)|\boldsymbol{W}), \boldsymbol{W}), \boldsymbol{W}), \boldsymbol{W})$$

- it should be clear that the gradients of this with respect to the weights can be found with the chain rule
- The back-propagated error will involve applying $\boldsymbol{f}$ multiple times
- Each time the error will get multiplied by some factor a
- If $\boldsymbol{y}(t)$ depends on the input $\boldsymbol{x}(t-\tau)$ then the back-propagated signal will be proportional to $a^{\tau-1}$
- This either vanishes or explodes when τ becomes large

Vanishing and Exploding Gradients

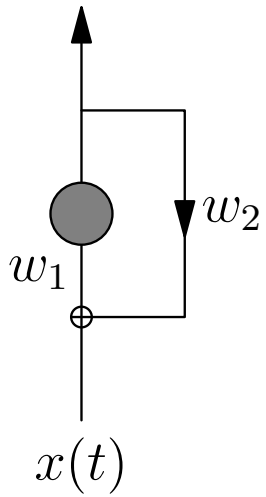
$$y(t) = w_1 \left(x(t) + w_2 \, y(t-1) \right)$$



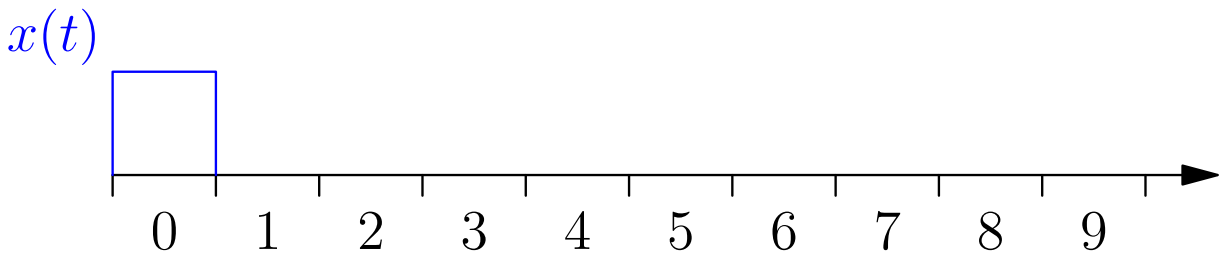


Vanishing and Exploding Gradients

$$y(t) = w_1 \left(x(t) + w_2 \, y(t-1) \right)$$

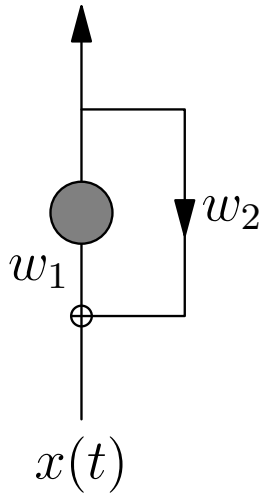


$$w_1 = w_2 = 0.9$$

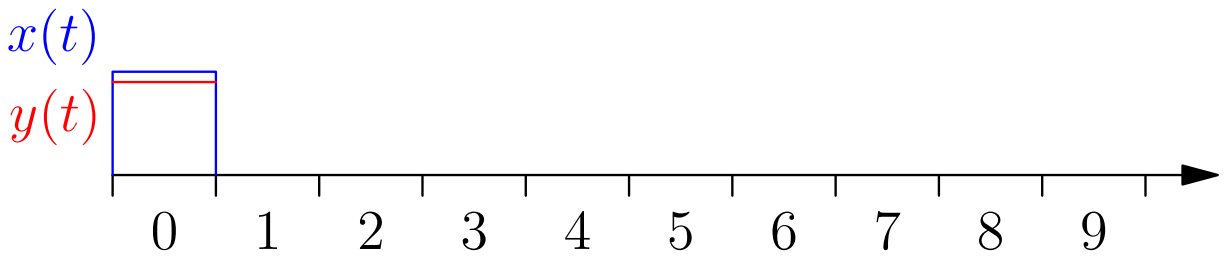


Vanishing and Exploding Gradients

$$y(t) = w_1 \left(x(t) + w_2 \, y(t-1) \right)$$

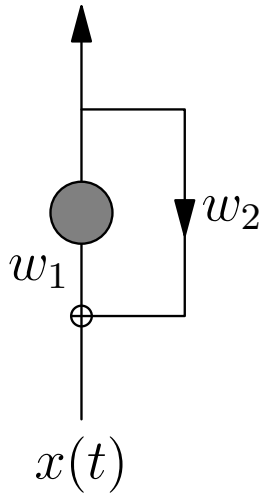


$$w_1 = w_2 = 0.9$$

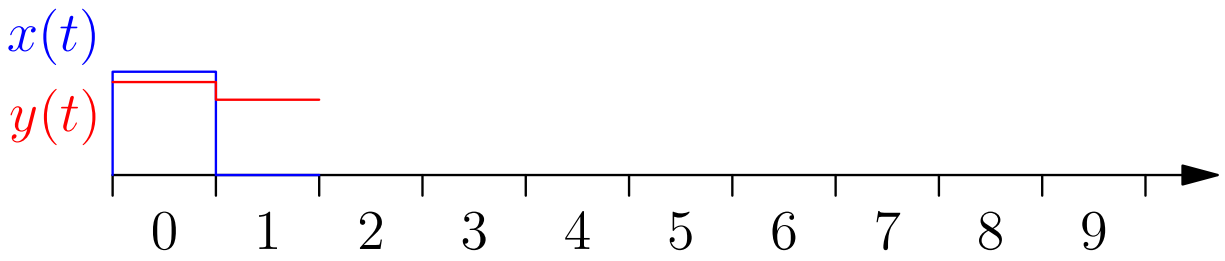


Vanishing and Exploding Gradients

$$y(t) = w_1 \left(x(t) + w_2 \, y(t-1) \right)$$

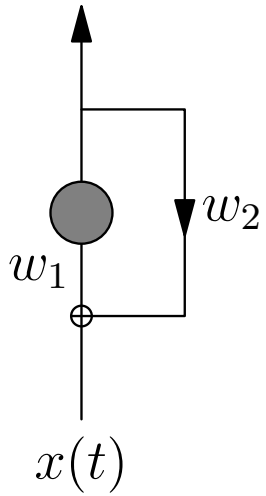


$$w_1 = w_2 = 0.9$$

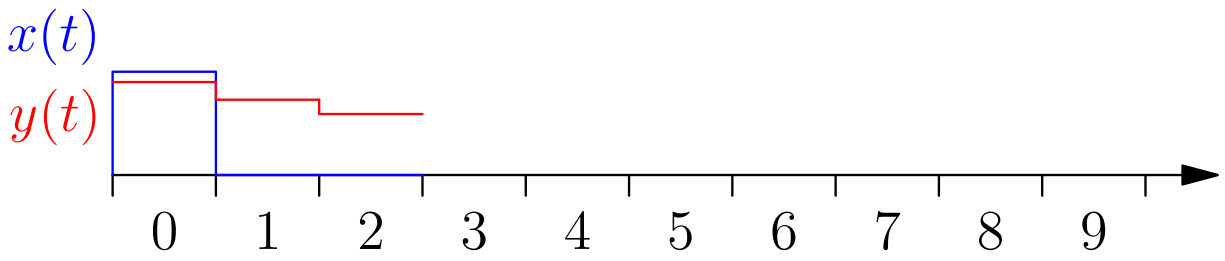


Vanishing and Exploding Gradients

$$y(t) = w_1 \left(x(t) + w_2 \, y(t-1) \right)$$

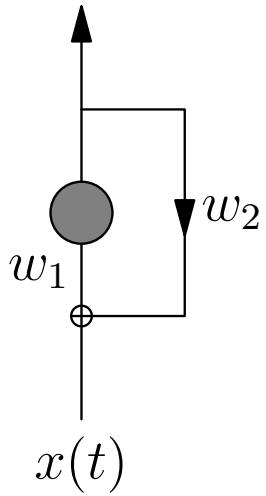


$$w_1 = w_2 = 0.9$$

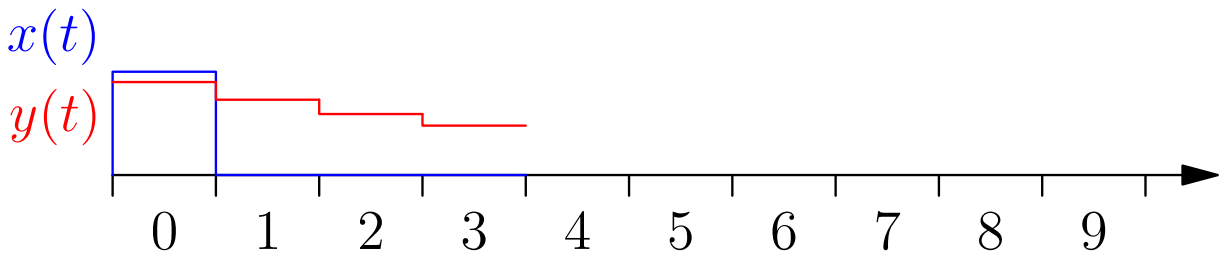


Vanishing and Exploding Gradients

$$y(t) = w_1 \left(x(t) + w_2 \, y(t-1) \right)$$

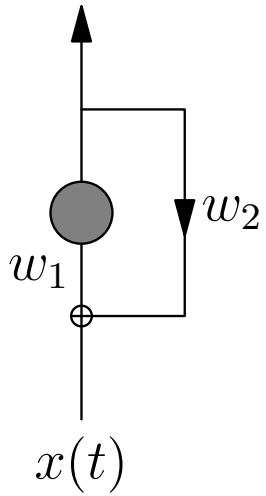


$$w_1 = w_2 = 0.9$$

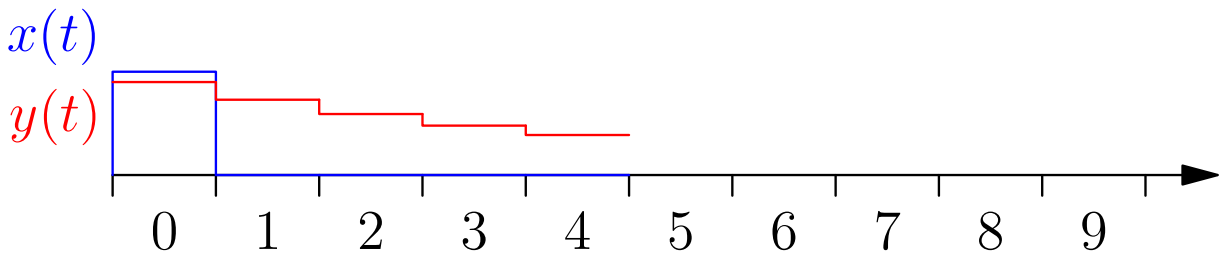


Vanishing and Exploding Gradients

$$y(t) = w_1 \left(x(t) + w_2 \, y(t-1) \right)$$

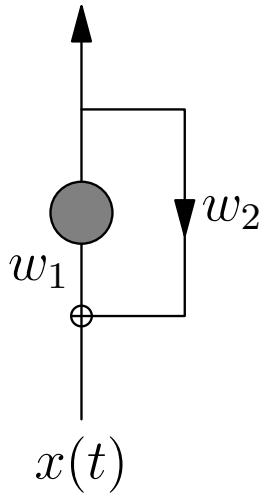


$$w_1 = w_2 = 0.9$$

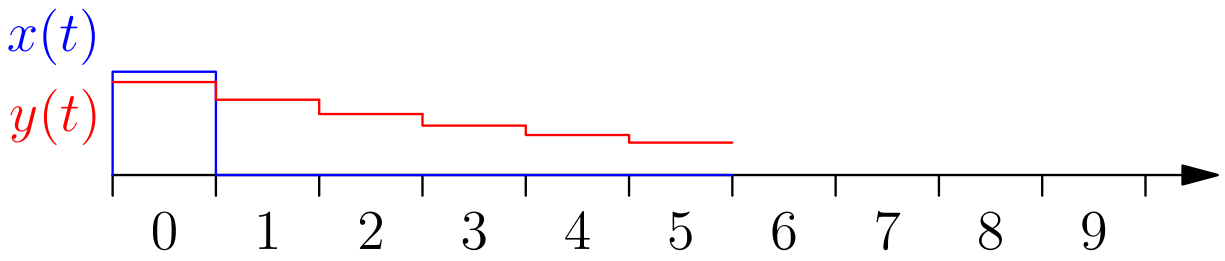


Vanishing and Exploding Gradients

$$y(t) = w_1 \left(x(t) + w_2 \, y(t-1) \right)$$

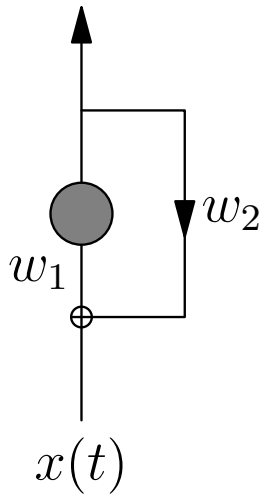


$$w_1 = w_2 = 0.9$$

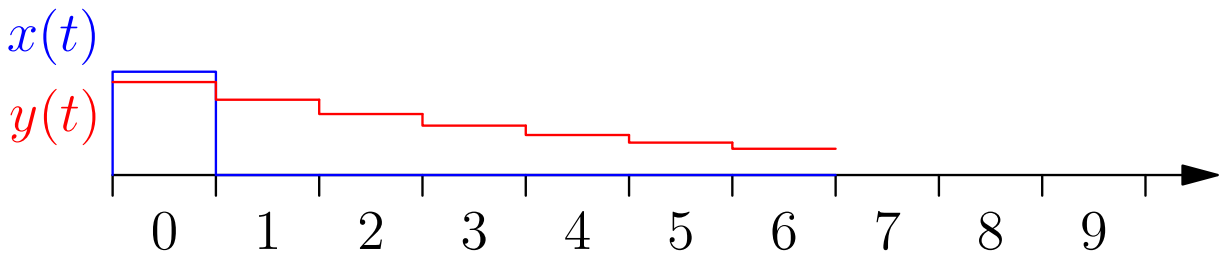


Vanishing and Exploding Gradients

$$y(t) = w_1 \left(x(t) + w_2 \, y(t-1) \right)$$

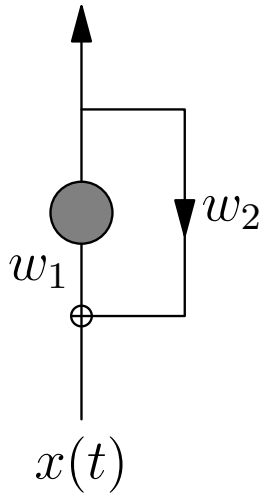


$$w_1 = w_2 = 0.9$$

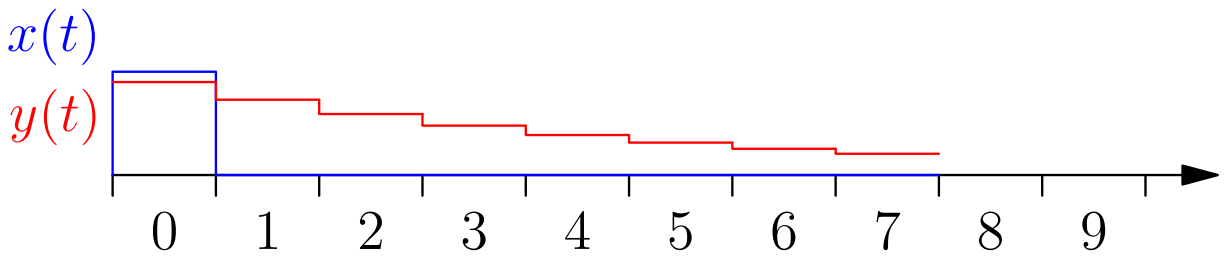


Vanishing and Exploding Gradients

$$y(t) = w_1 \left(x(t) + w_2 \, y(t-1) \right)$$

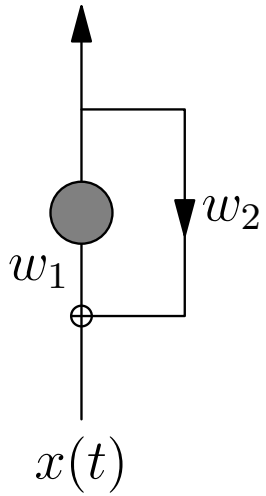


$$w_1 = w_2 = 0.9$$

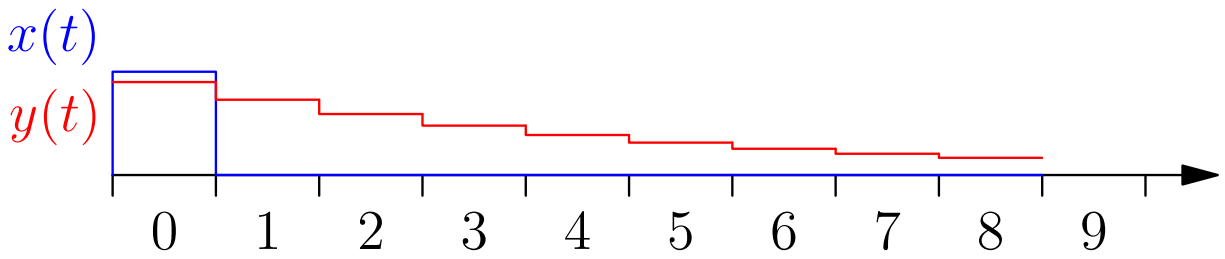


Vanishing and Exploding Gradients

$$y(t) = w_1 \left(x(t) + w_2 \, y(t-1) \right)$$

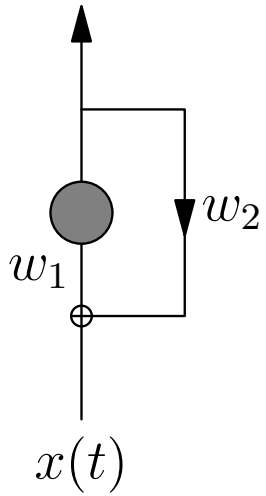


$$w_1 = w_2 = 0.9$$

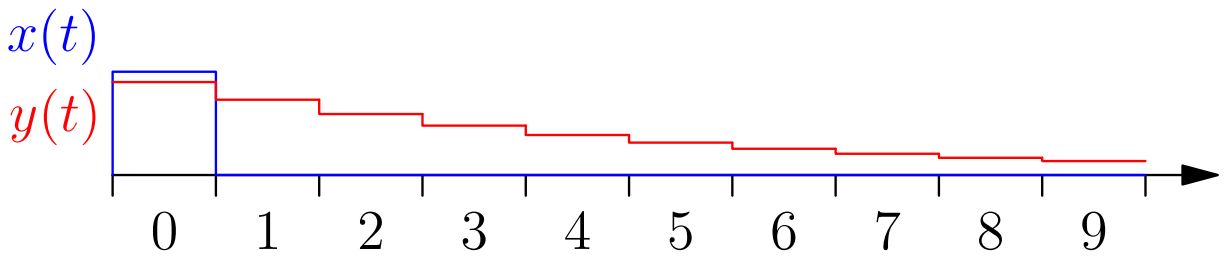


Vanishing and Exploding Gradients

$$y(t) = w_1 \left(x(t) + w_2 \, y(t-1) \right)$$

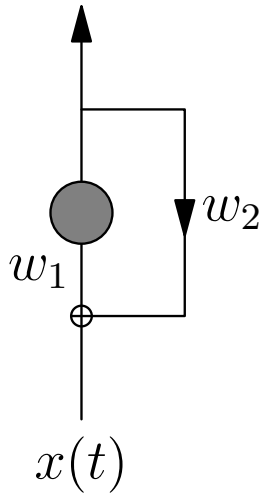


$$w_1 = w_2 = 0.9$$

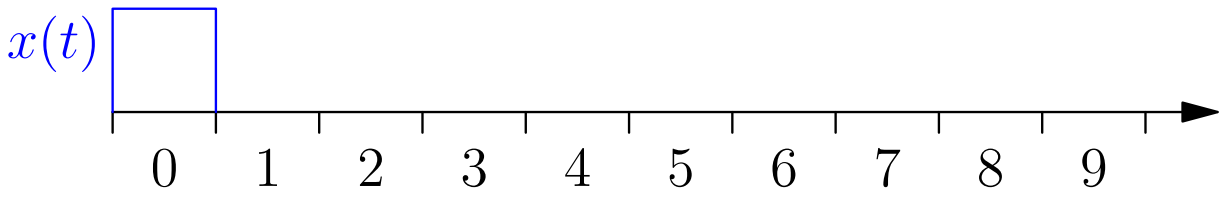


Vanishing and Exploding Gradients

$$y(t) = w_1 \left(x(t) + w_2 \, y(t-1) \right)$$

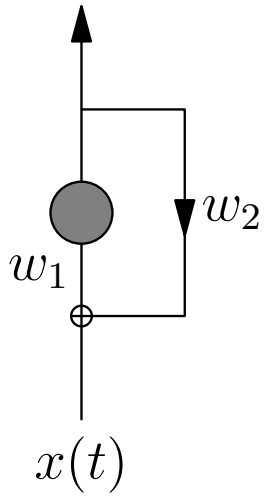


$$w_1 = w_2 = 1.1$$

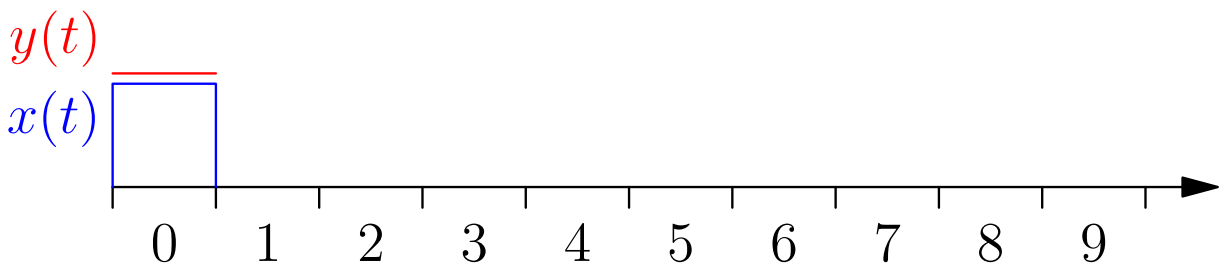


Vanishing and Exploding Gradients

$$y(t) = w_1 \left(x(t) + w_2 \, y(t-1) \right)$$

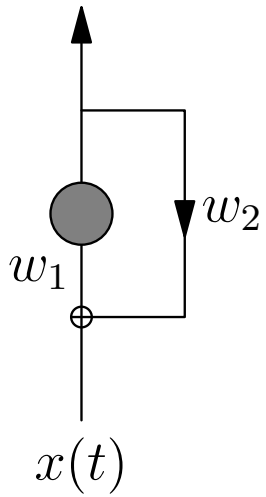


$$w_1 = w_2 = 1.1$$

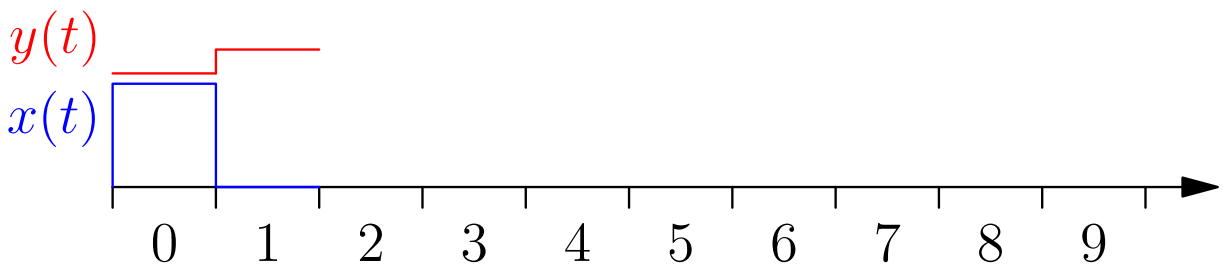


Vanishing and Exploding Gradients

$$y(t) = w_1 \left(x(t) + w_2 \, y(t-1) \right)$$

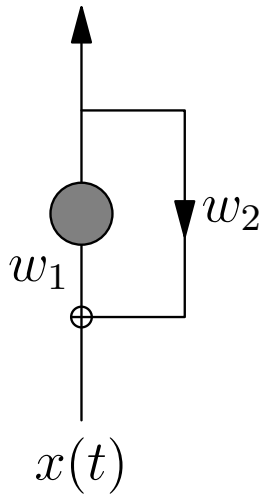


$$w_1 = w_2 = 1.1$$

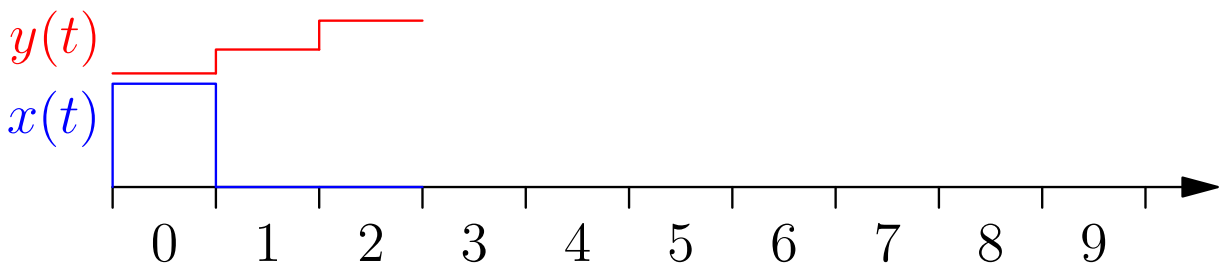


Vanishing and Exploding Gradients

$$y(t) = w_1 \left(x(t) + w_2 \, y(t-1) \right)$$

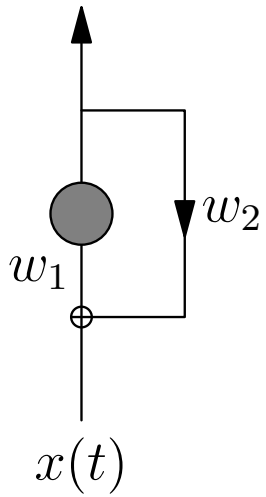


$$w_1 = w_2 = 1.1$$

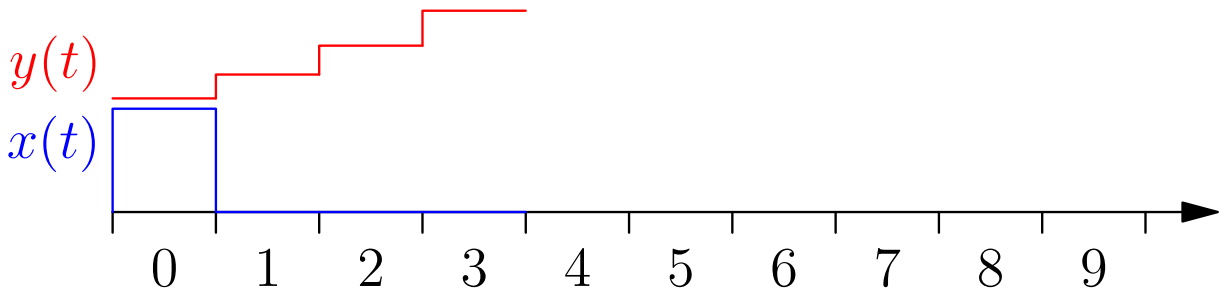


Vanishing and Exploding Gradients

$$y(t) = w_1 \left(x(t) + w_2 \, y(t-1) \right)$$

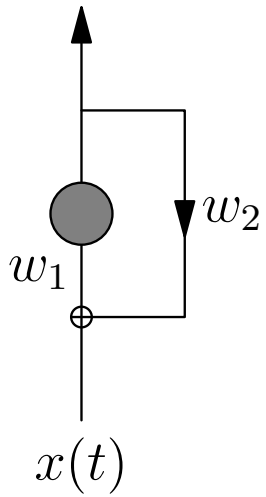


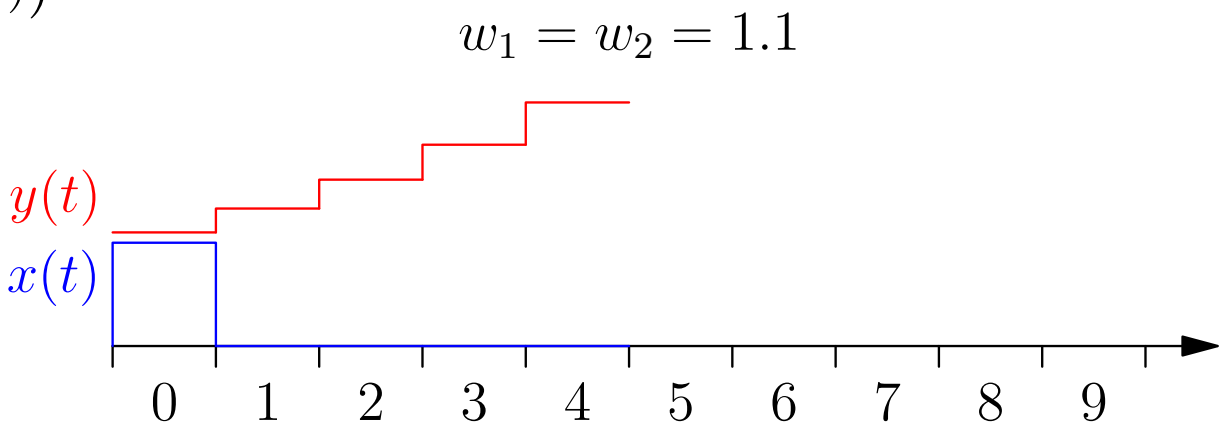
$$w_1 = w_2 = 1.1$$



Vanishing and Exploding Gradients

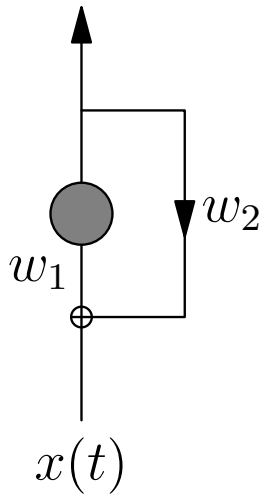
$$y(t) = w_1 \left(x(t) + w_2 \, y(t-1) \right)$$

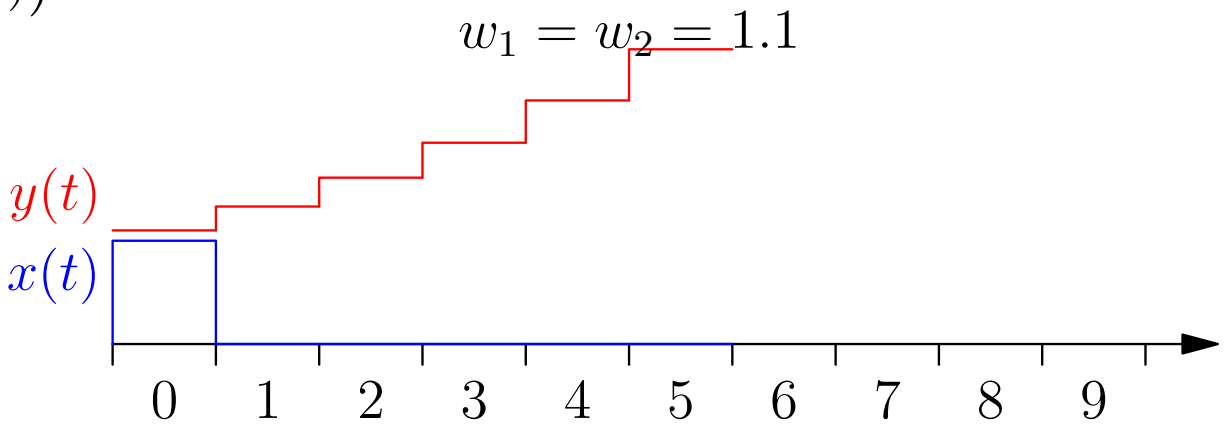




Vanishing and Exploding Gradients

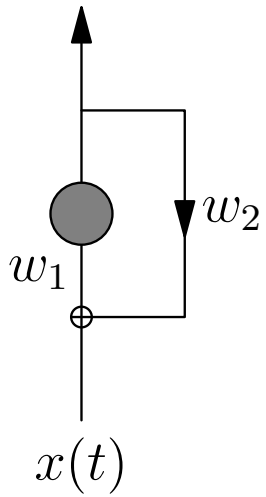
$$y(t) = w_1 \left(x(t) + w_2 \, y(t-1) \right)$$

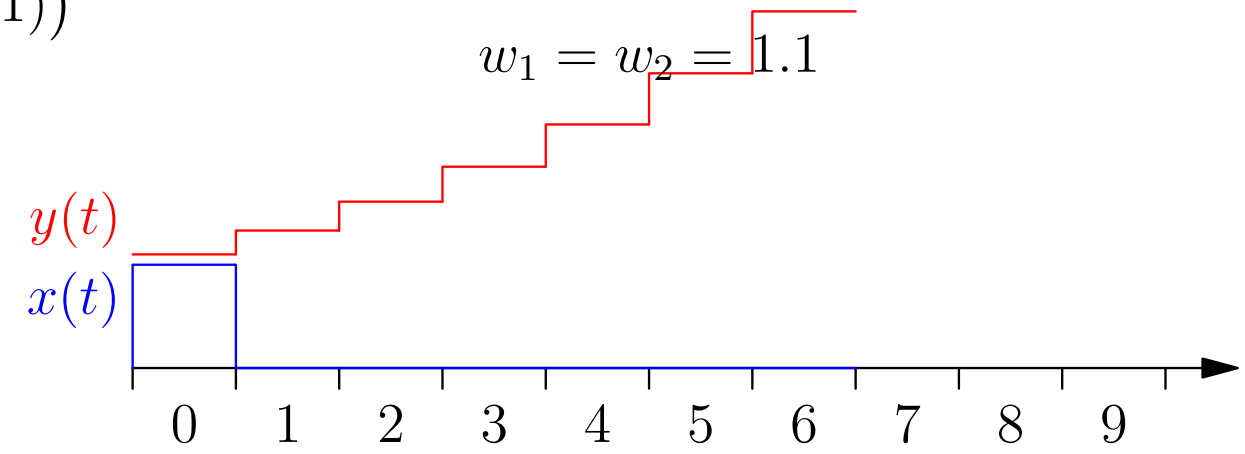




Vanishing and Exploding Gradients

$$y(t) = w_1 \left(x(t) + w_2 \, y(t-1) \right)$$





LSTM Architecture

- LSTMs (long-short term memory) was designed to solve this problem

LSTM Architecture

- LSTMs (long-short term memory) was designed to solve this problem
- Key ideas: to retain a 'long-term memory' requires

$$\boldsymbol{c}(t) = \boldsymbol{c}(t-1)$$

LSTM Architecture

- LSTMs (long-short term memory) was designed to solve this problem
- Key ideas: to retain a 'long-term memory' requires

$$\boldsymbol{c}(t) = \boldsymbol{c}(t-1)$$

- Sometimes we have to forget and sometimes we have to change a memory

LSTM Architecture

- LSTMs (long-short term memory) was designed to solve this problem
- Key ideas: to retain a 'long-term memory' requires

$$\boldsymbol{c}(t) = \boldsymbol{c}(t-1)$$

- Sometimes we have to forget and sometimes we have to change a memory
- To do this we should use 'gates' that saturate at 0 and 1

LSTM Architecture

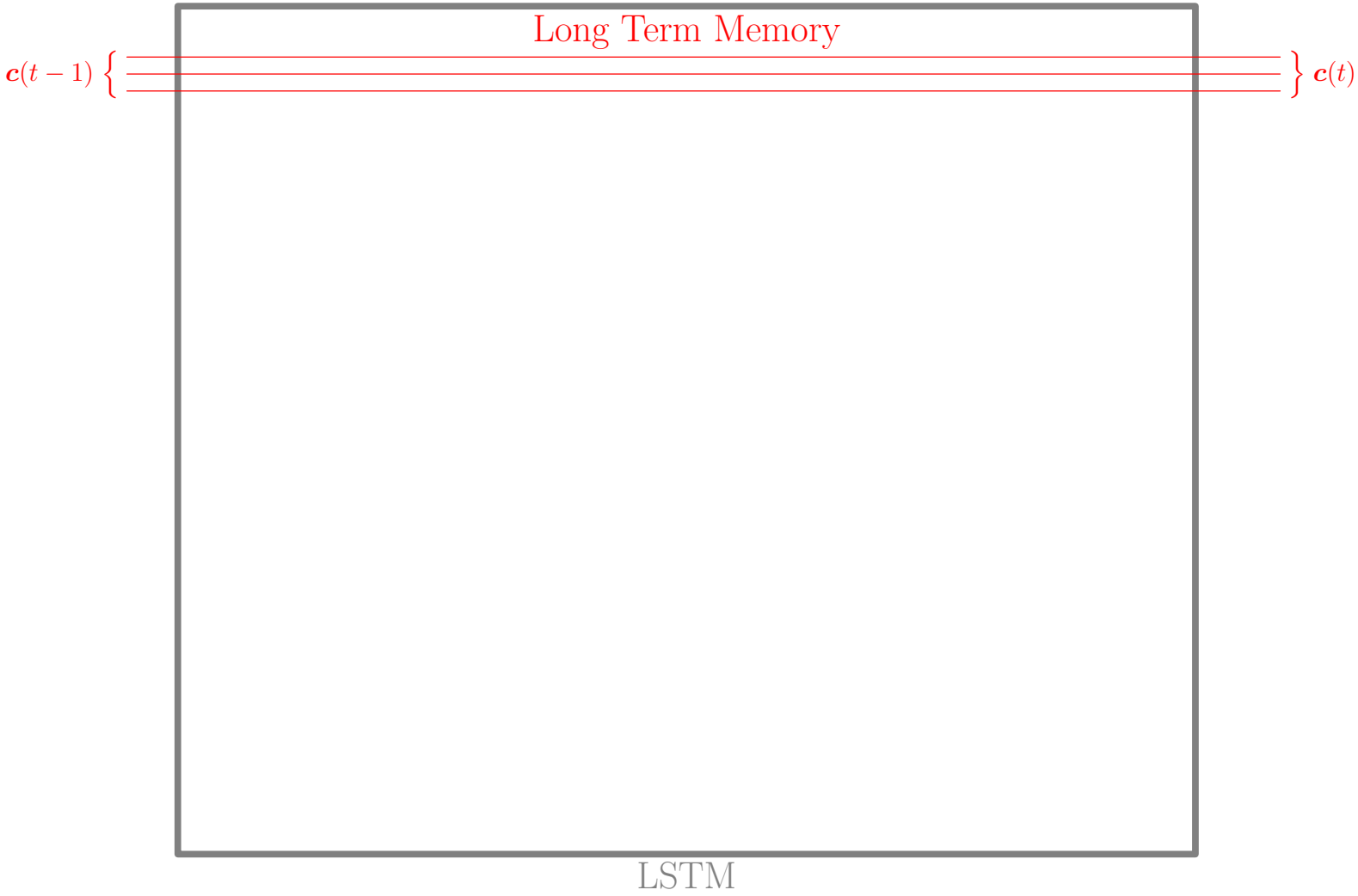
- LSTMs (long-short term memory) was designed to solve this problem
- Key ideas: to retain a 'long-term memory' requires

$$\boldsymbol{c}(t) = \boldsymbol{c}(t-1)$$

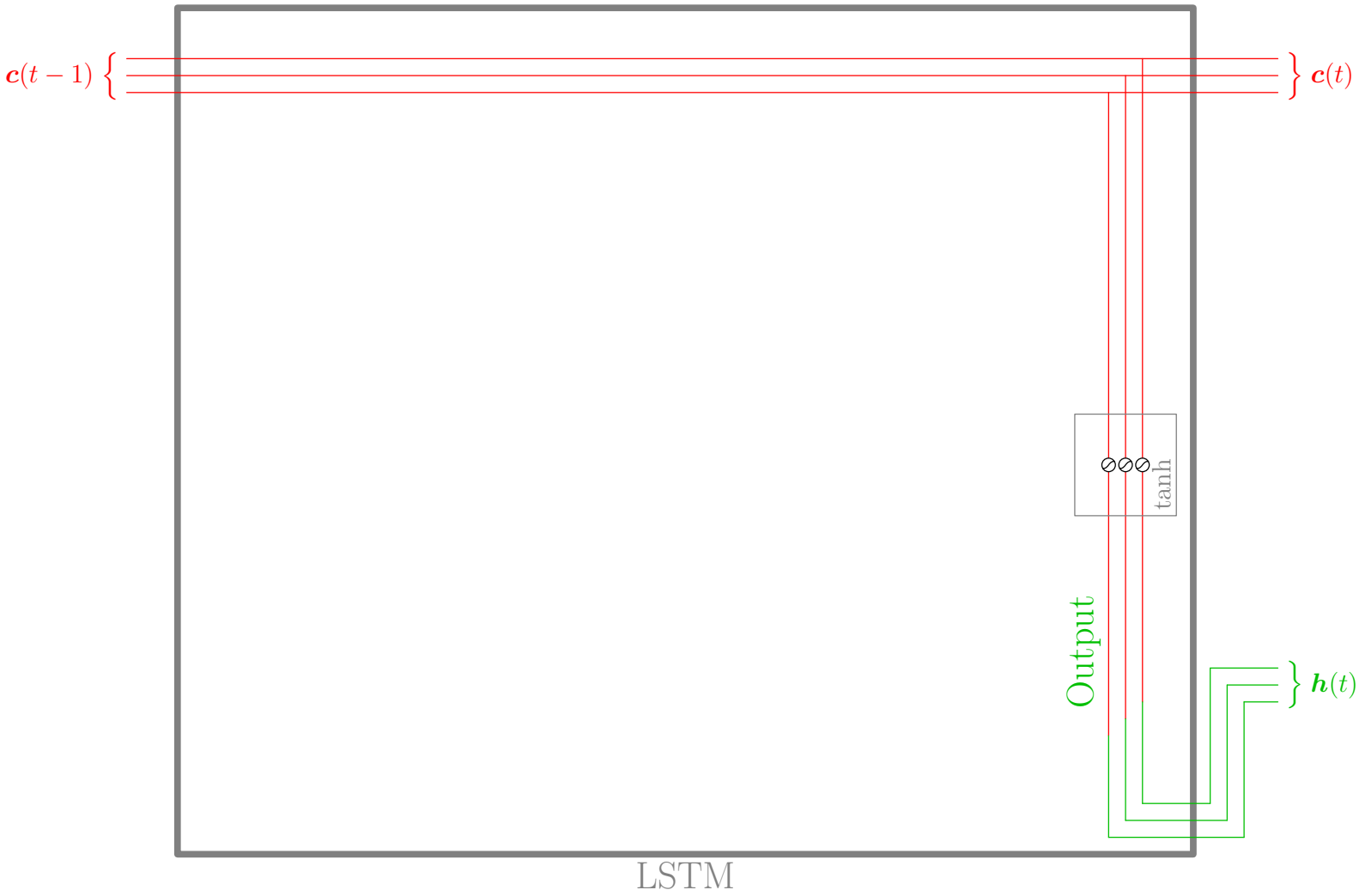
- Sometimes we have to forget and sometimes we have to change a memory
- To do this we should use 'gates' that saturate at 0 and 1
- Sigmoid functions naturally saturate at 0 and 1

LSTM Architecture

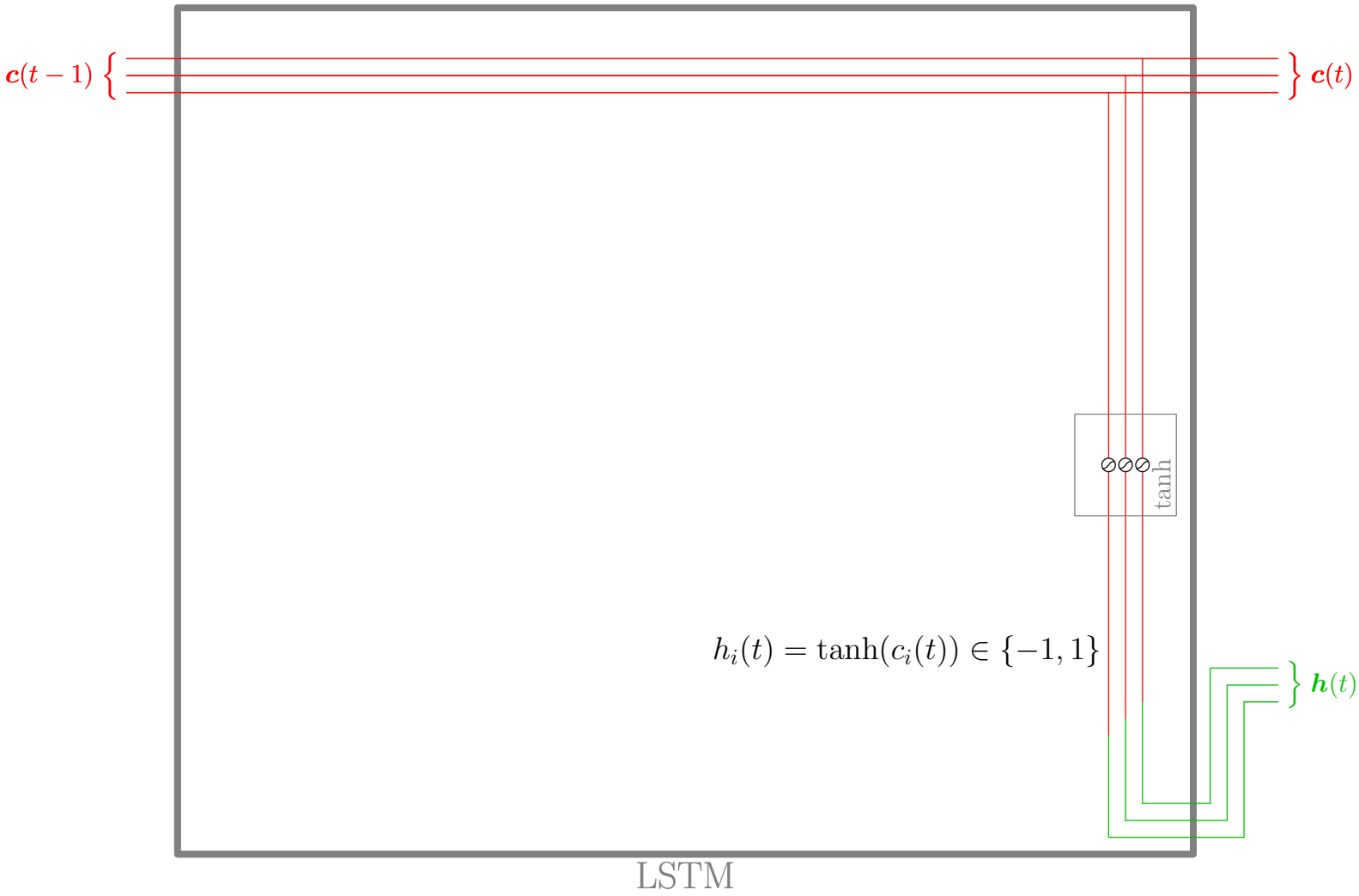
LSTM Architecture



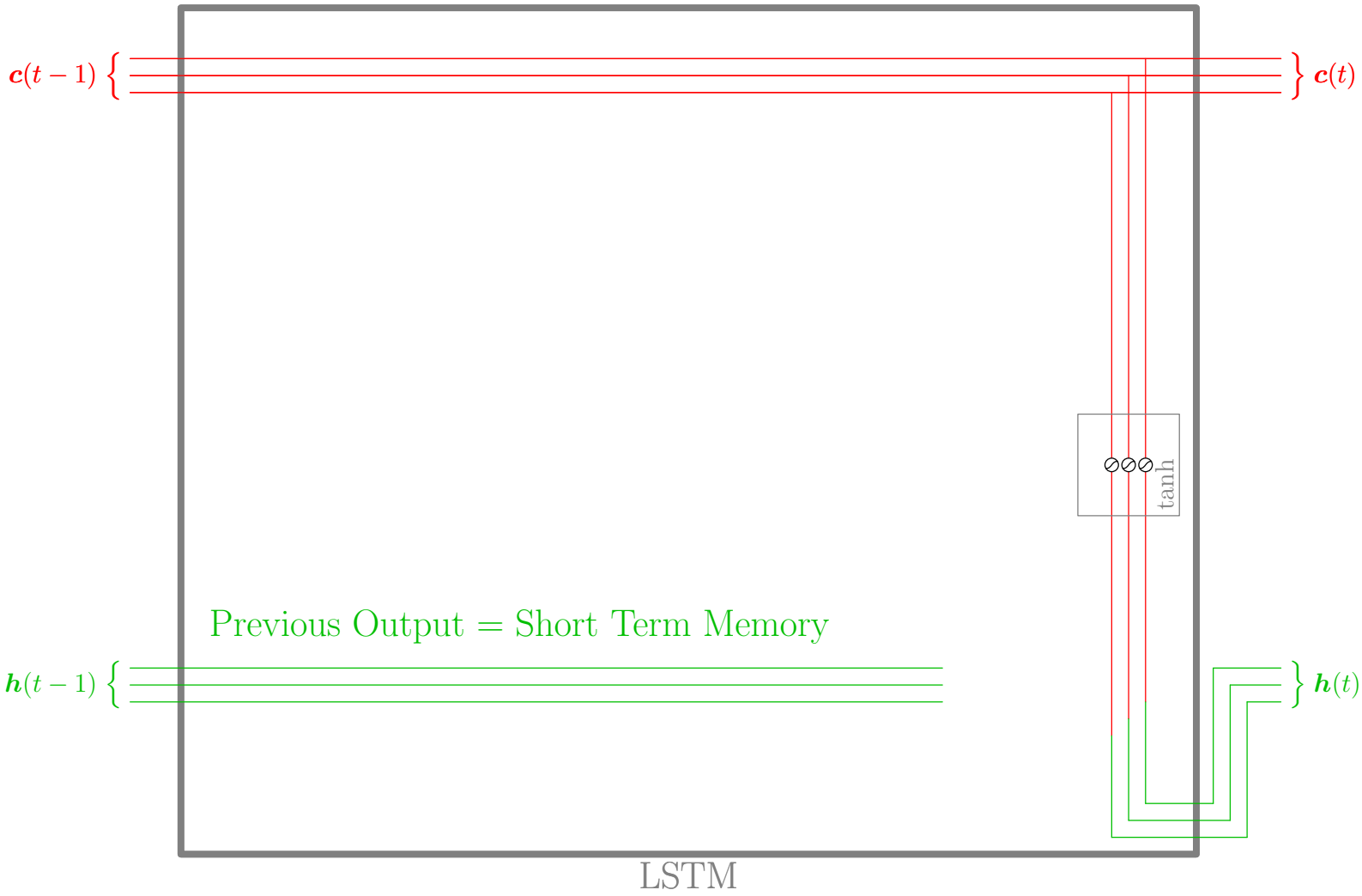
LSTM Architecture



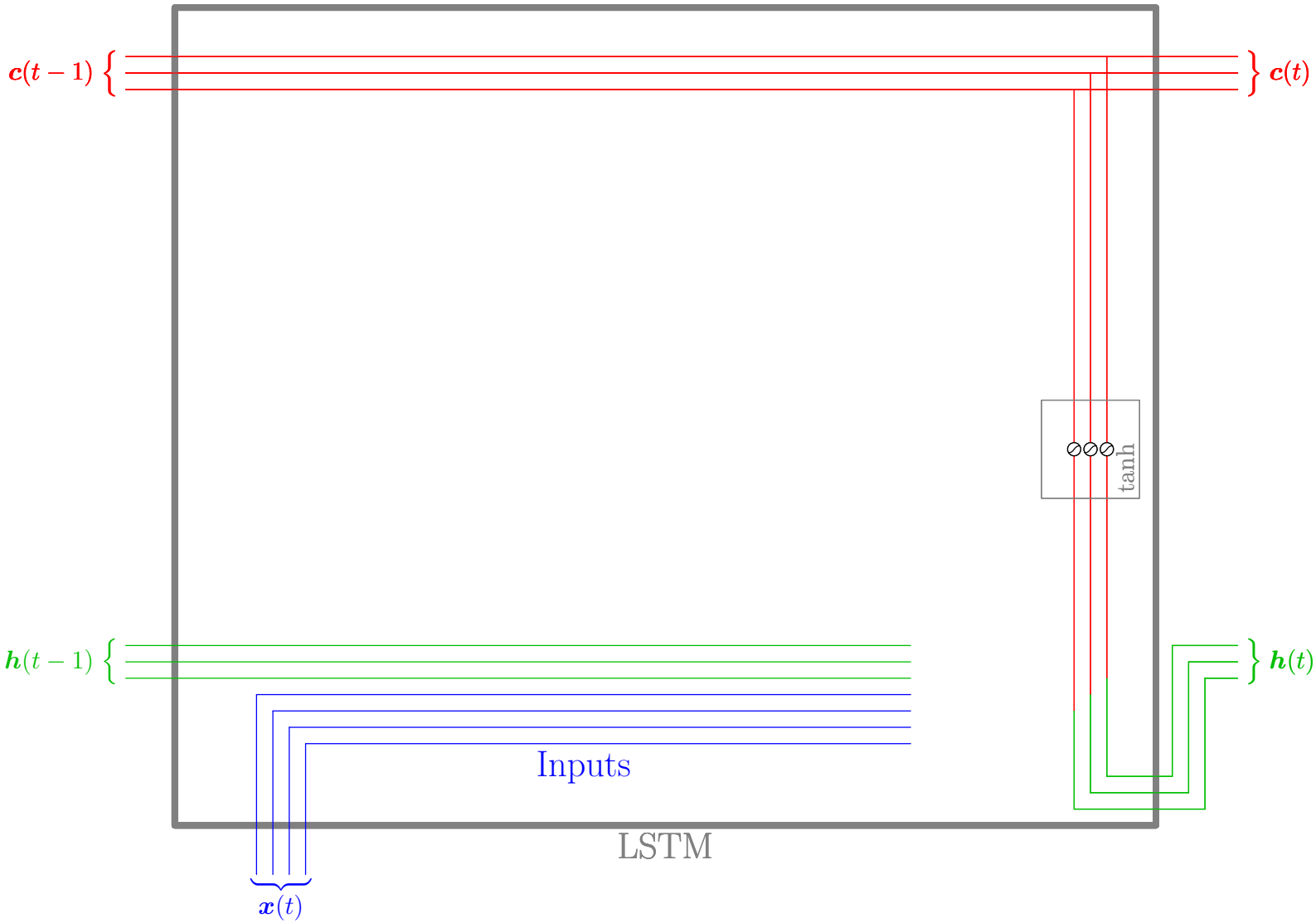
LSTM Architecture



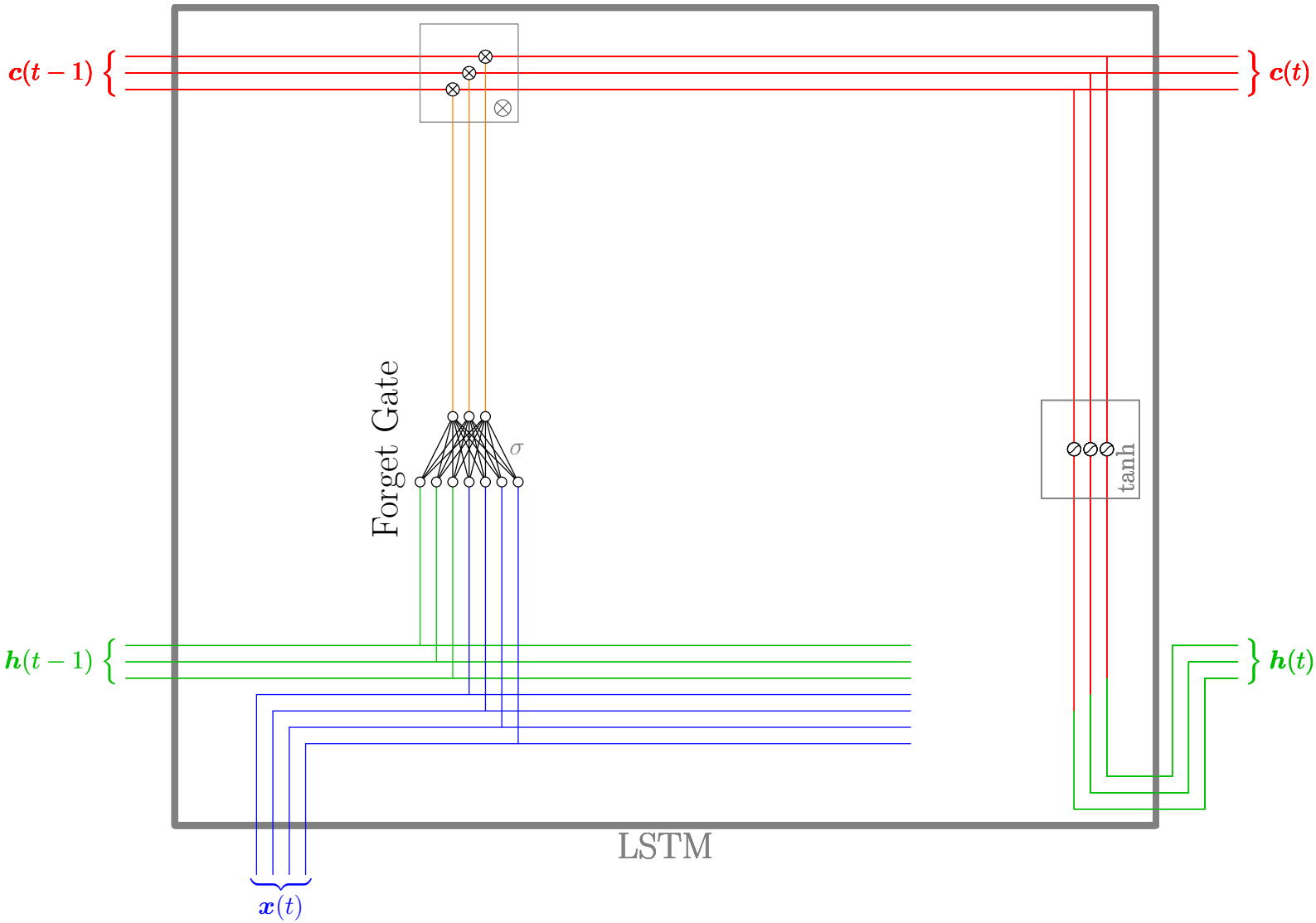
LSTM Architecture



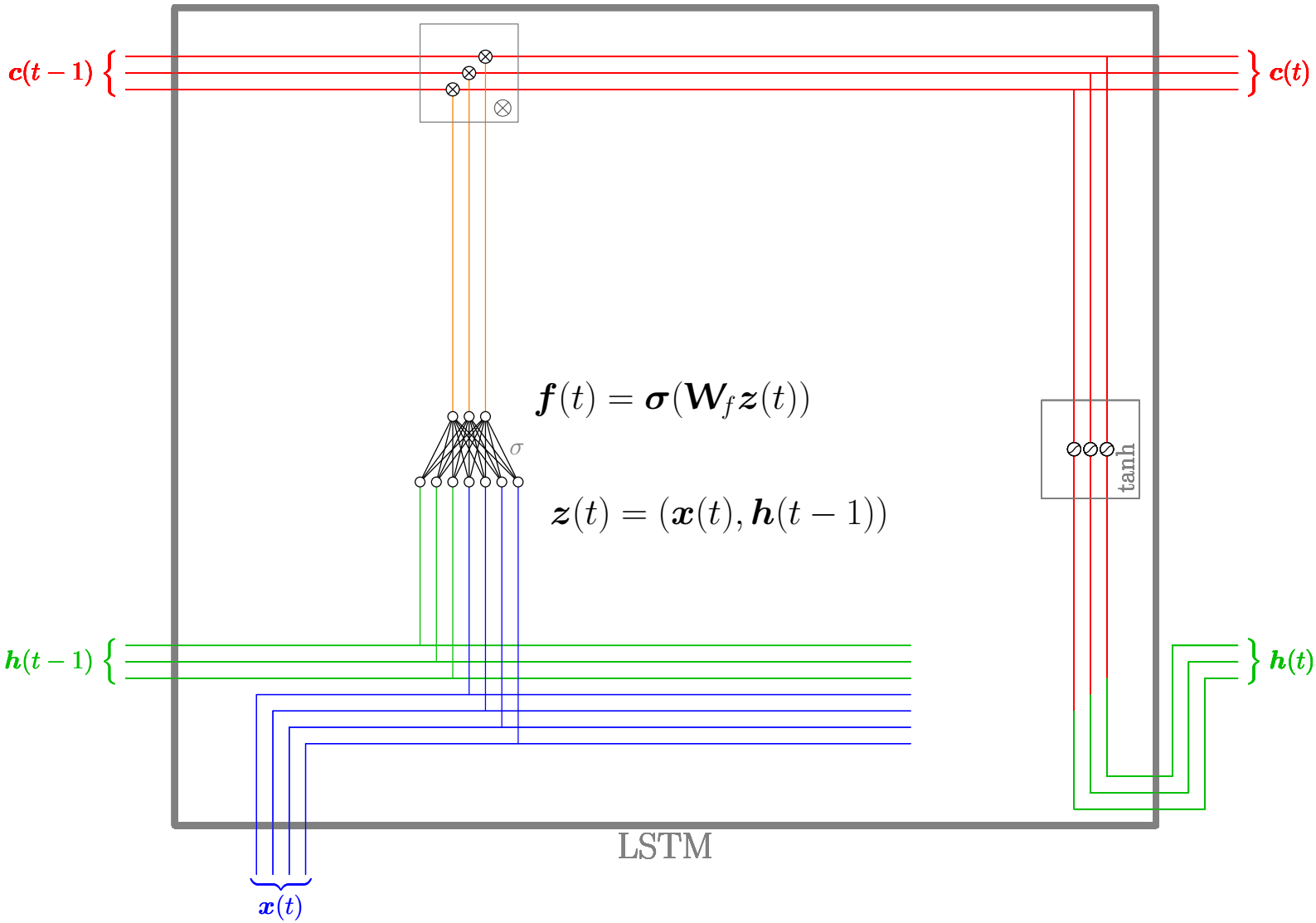
LSTM Architecture



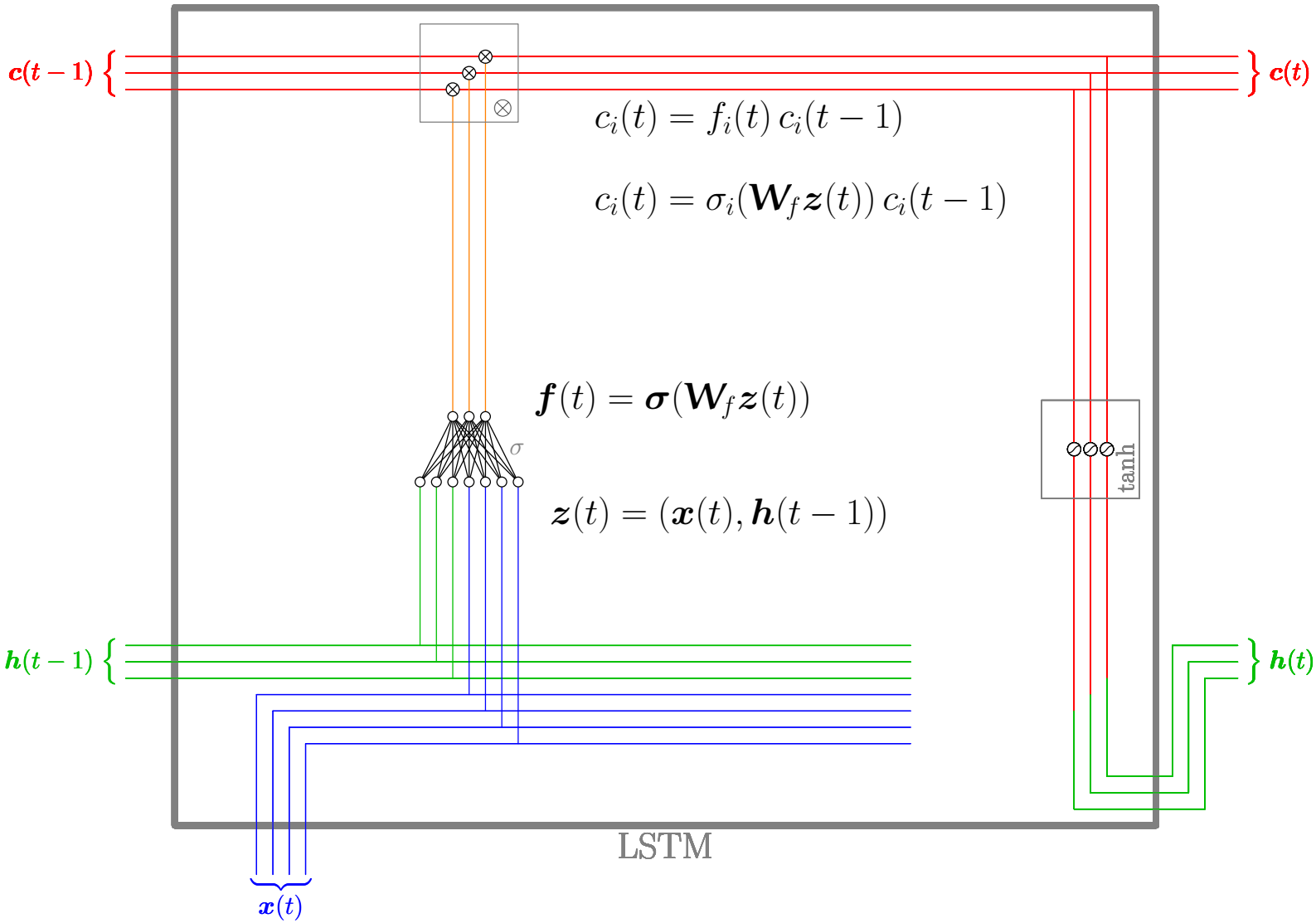
LSTM Architecture



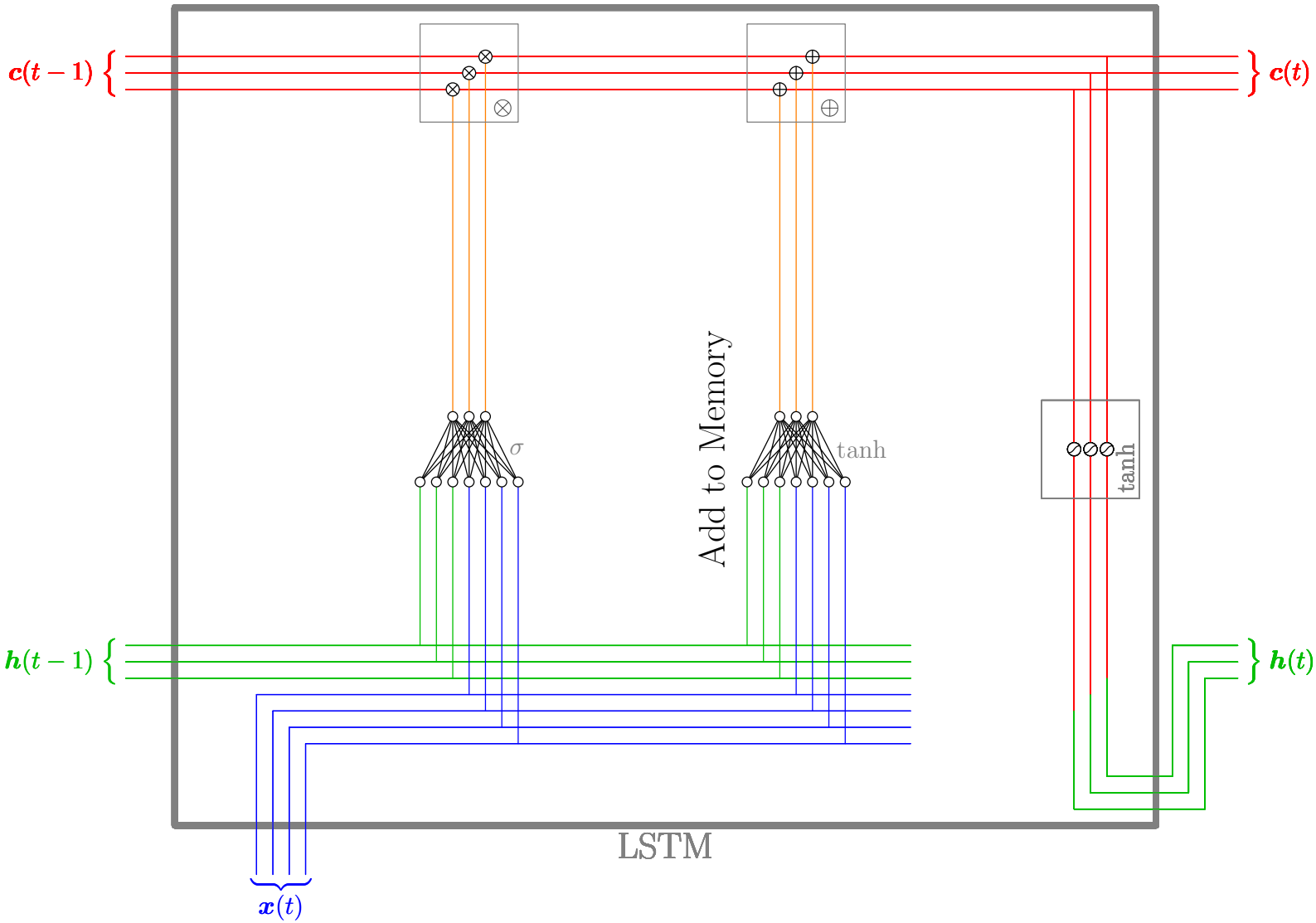
LSTM Architecture



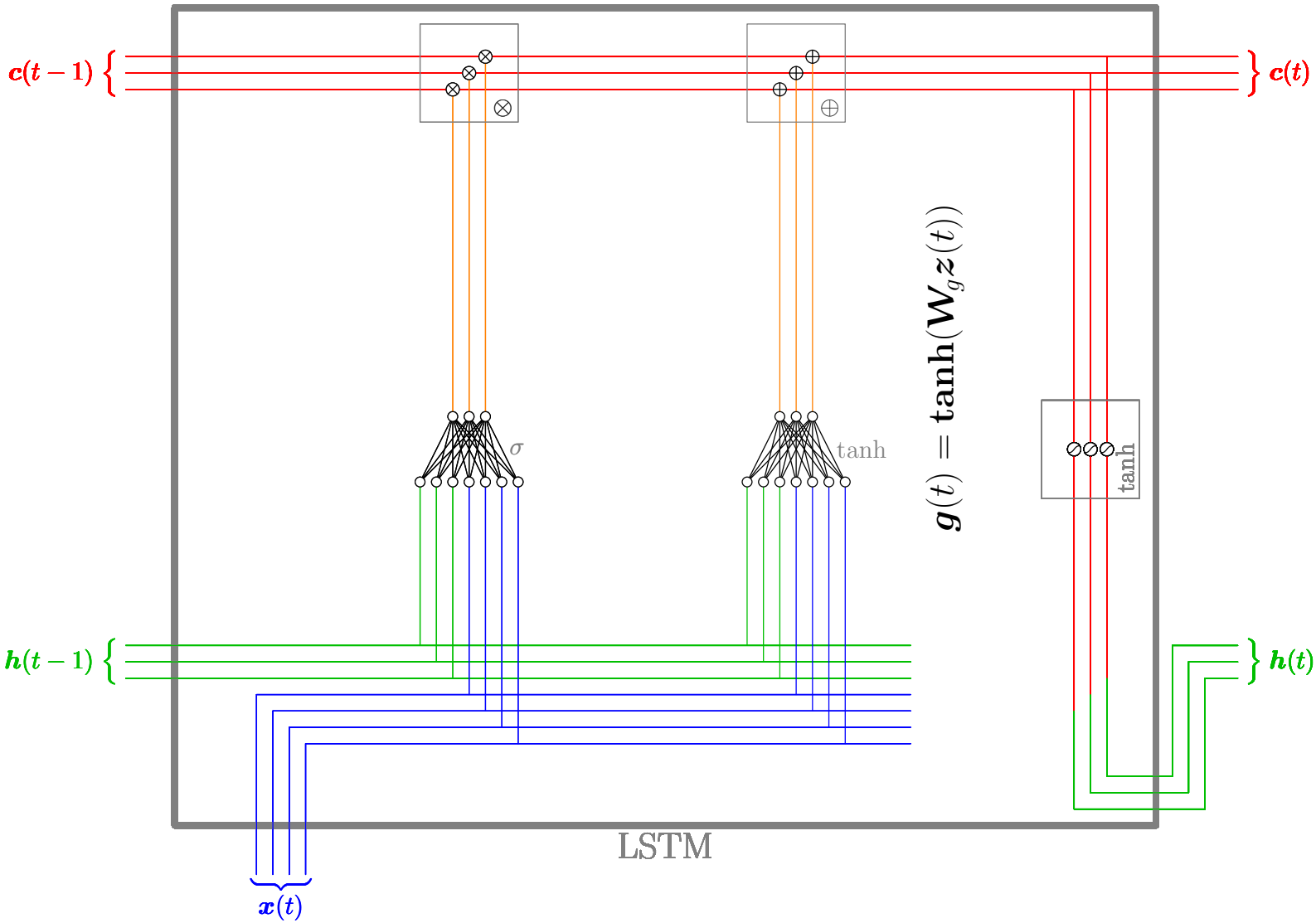
LSTM Architecture



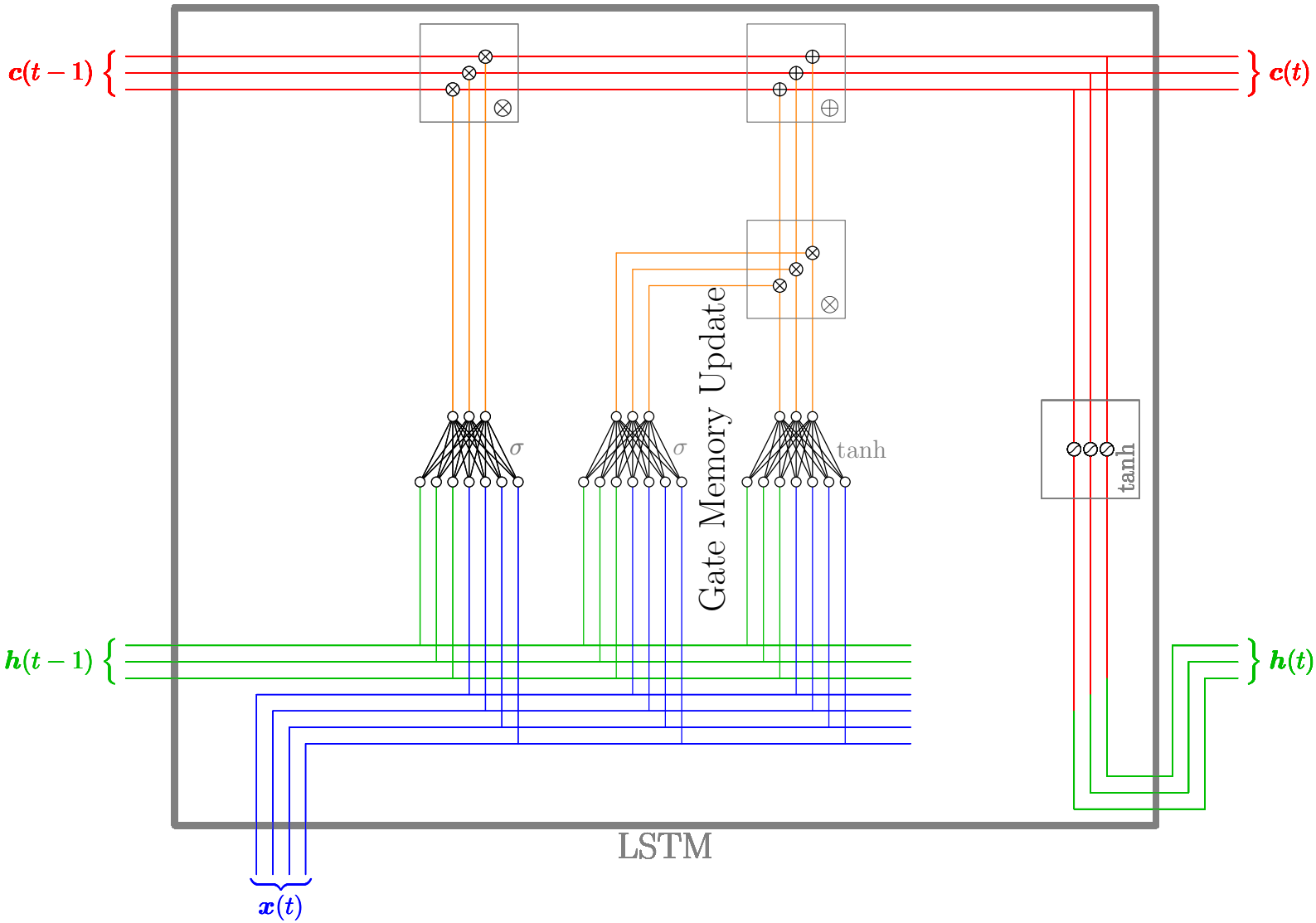
LSTM Architecture



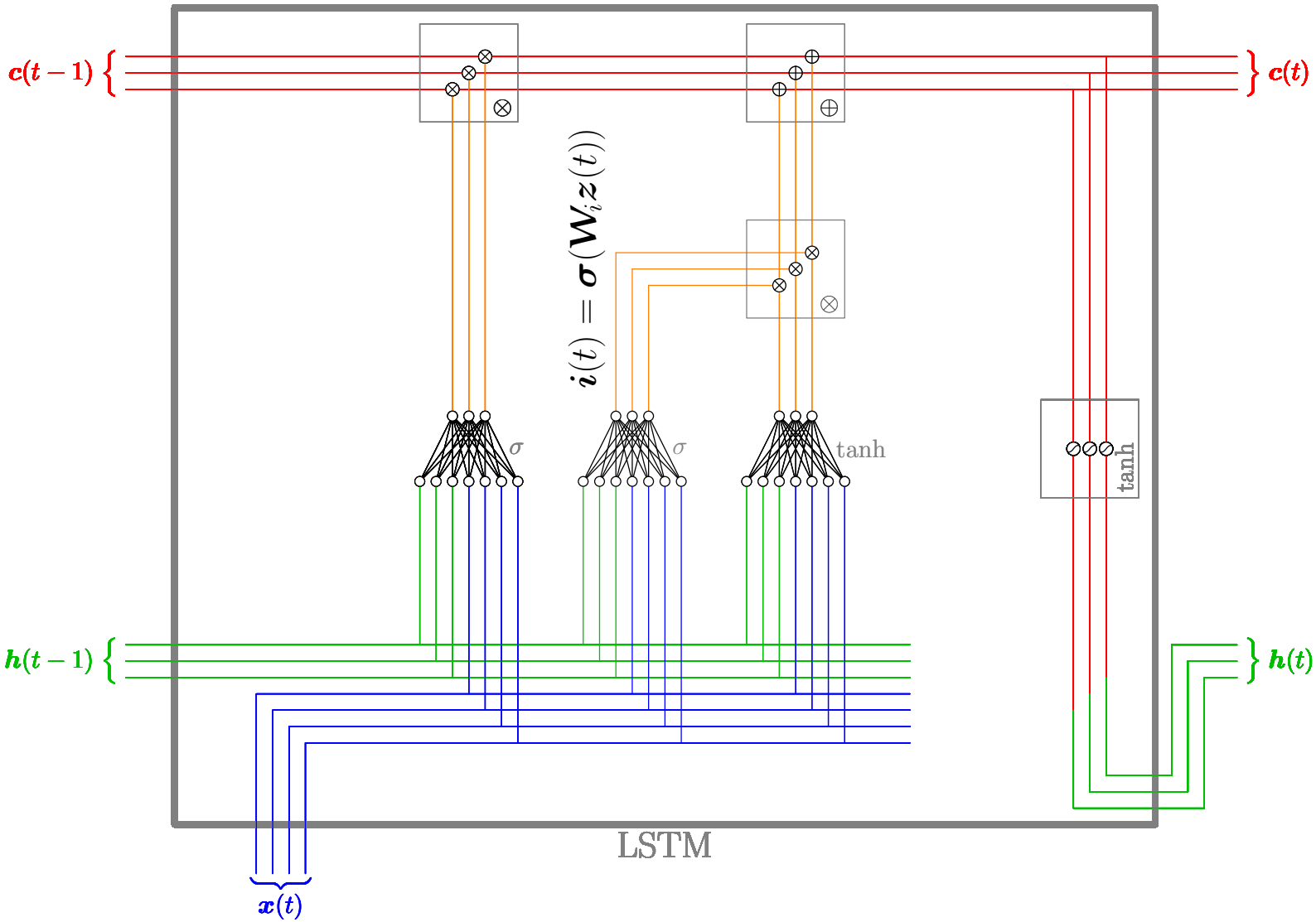
LSTM Architecture



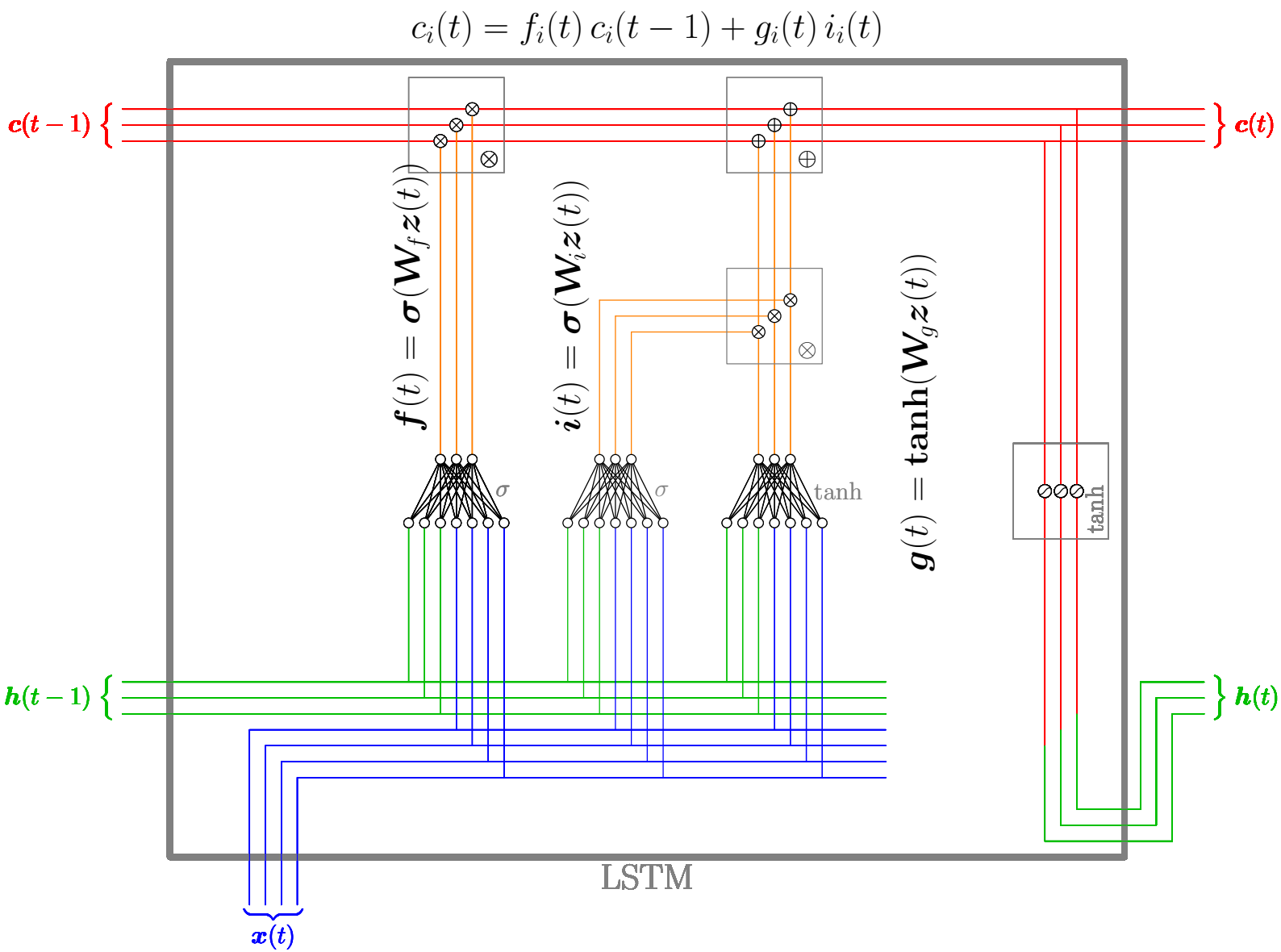
LSTM Architecture



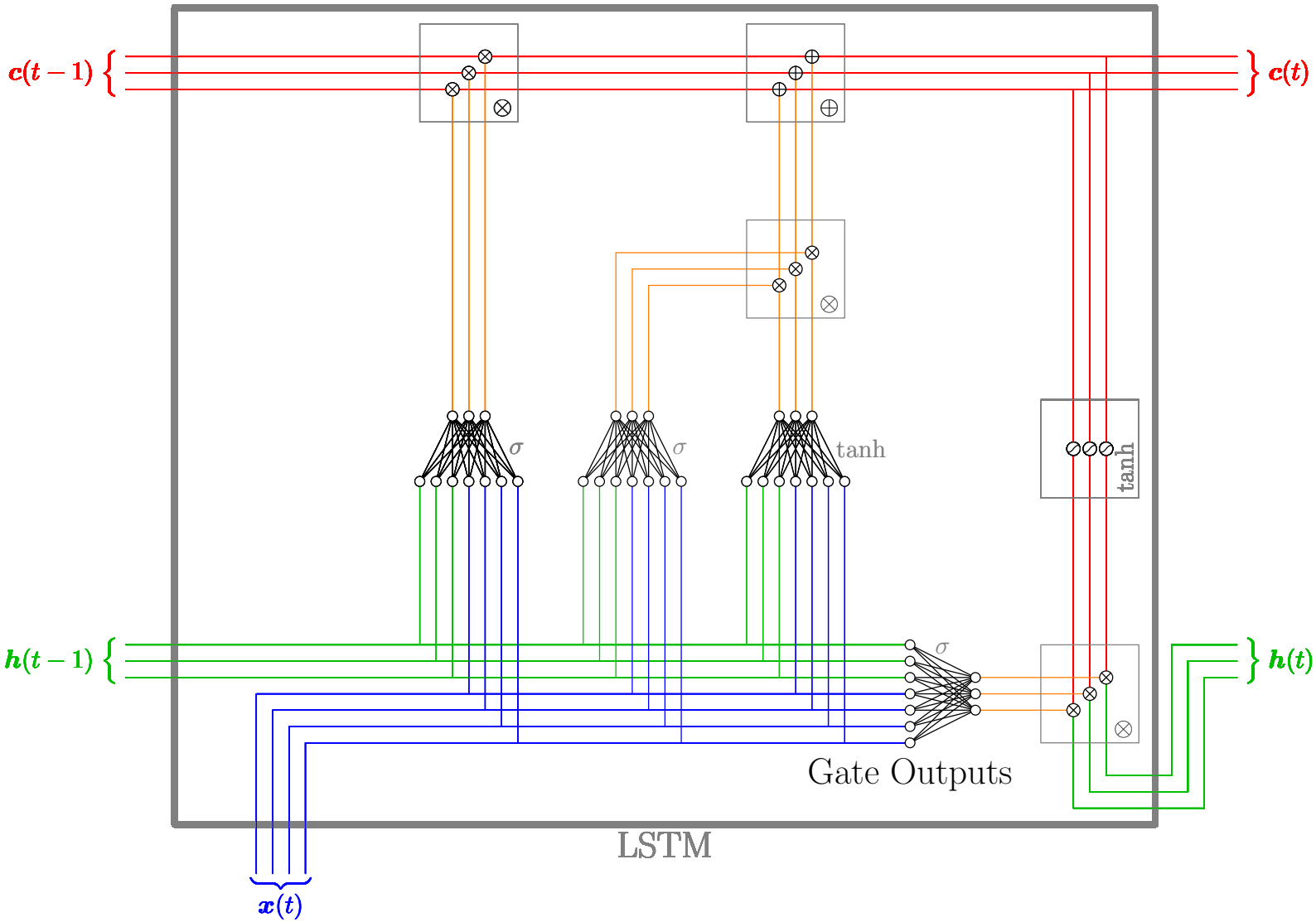
LSTM Architecture



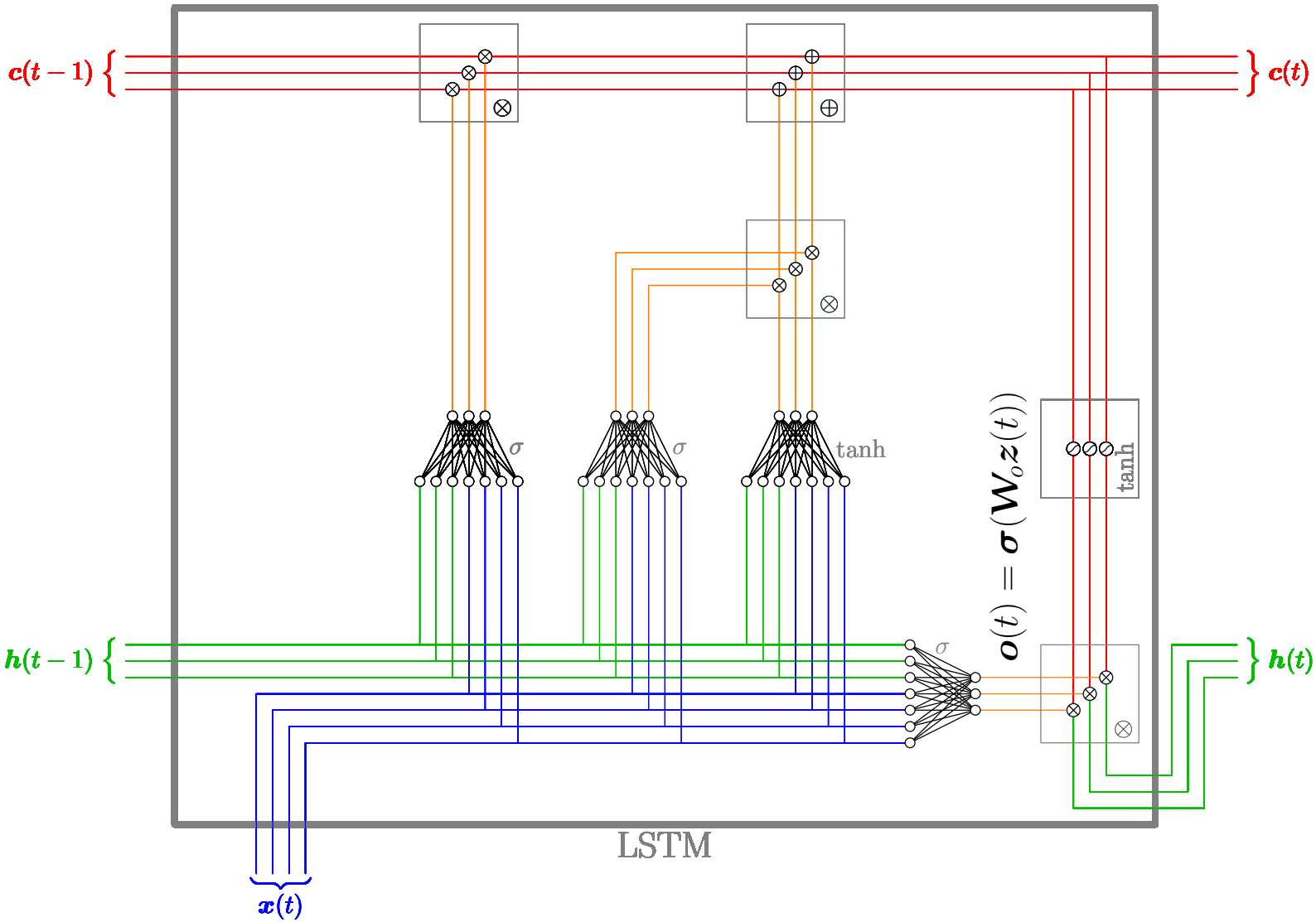
LSTM Architecture



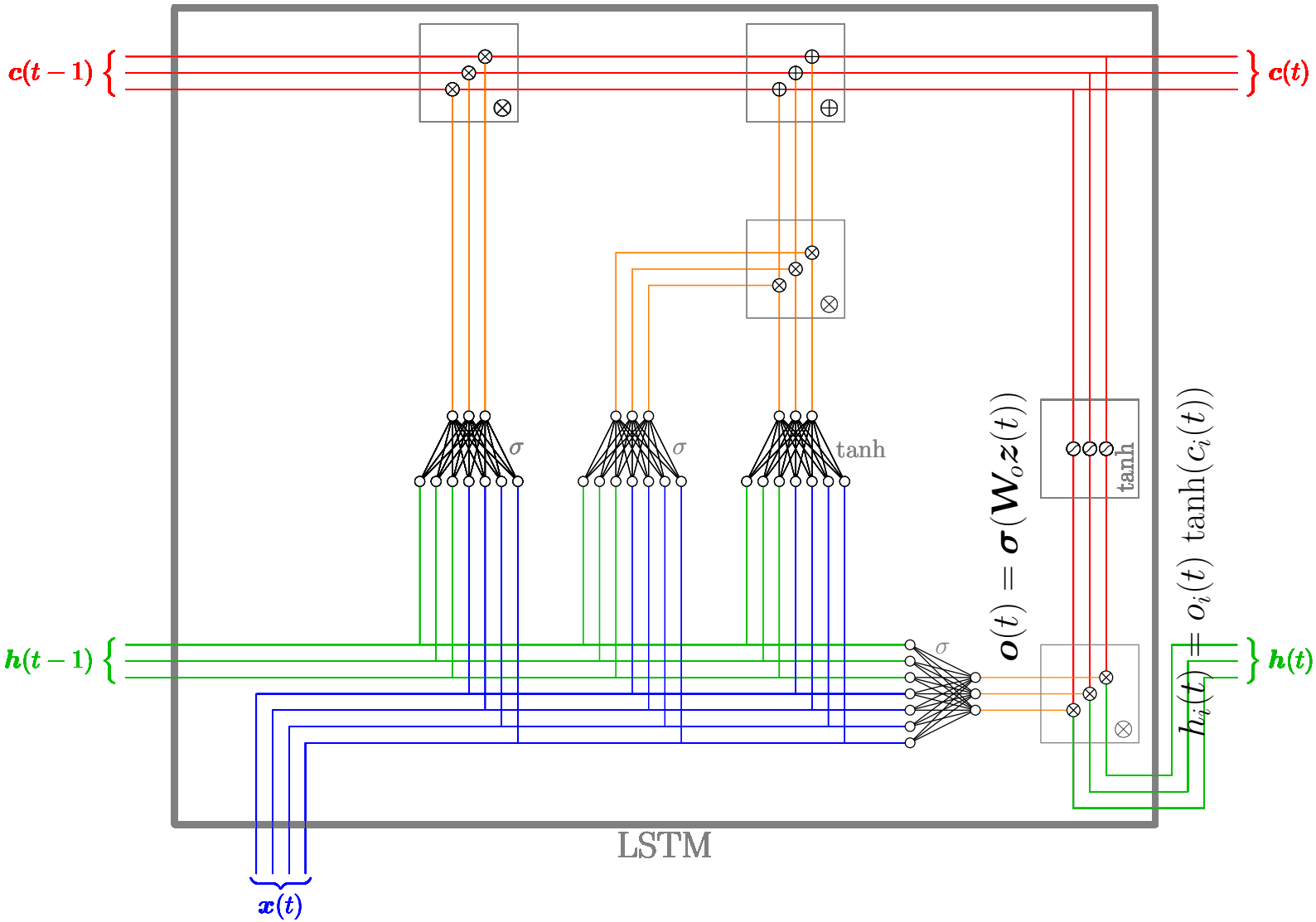
LSTM Architecture



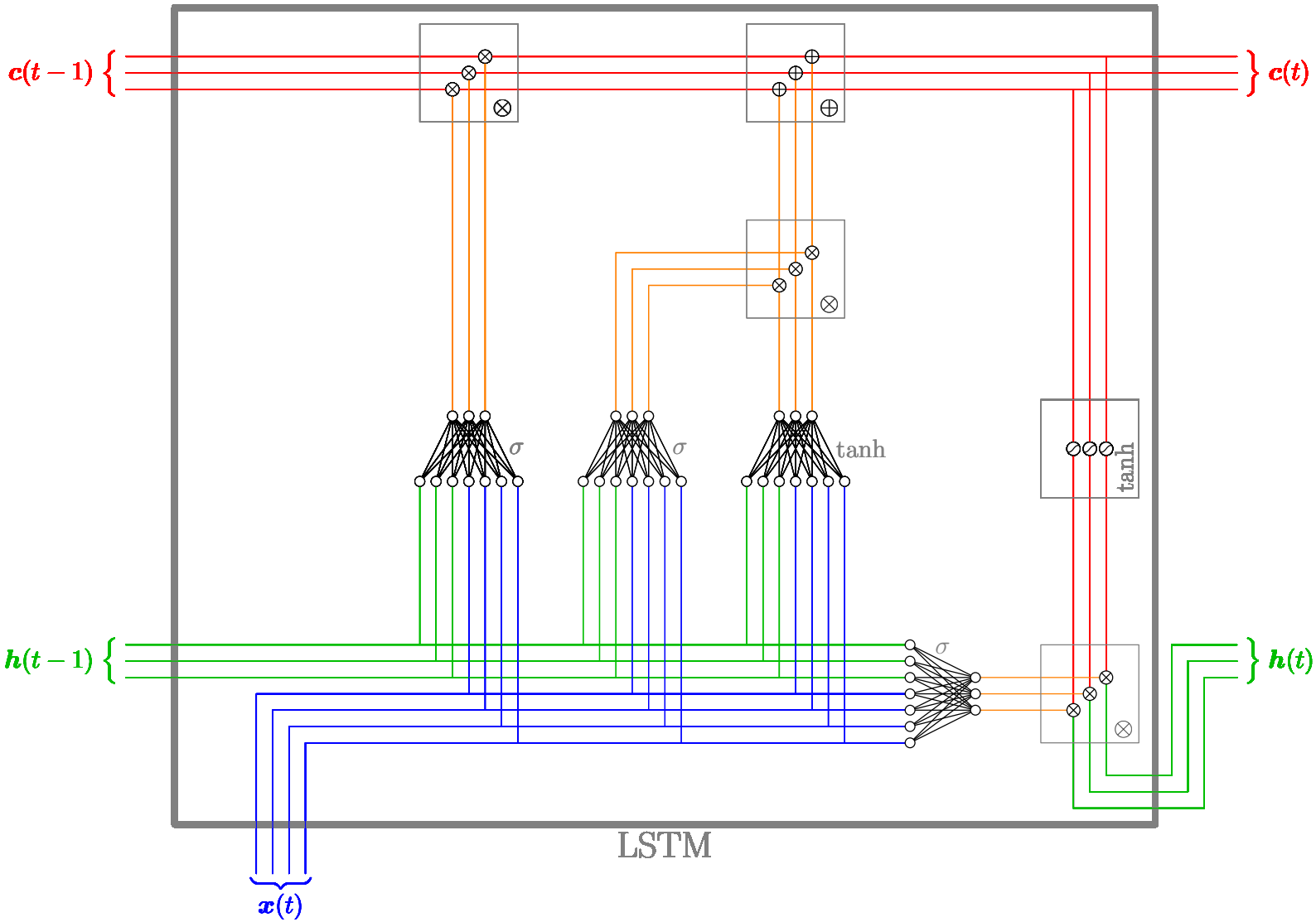
LSTM Architecture



LSTM Architecture



LSTM Architecture



Update Equations

- Inputs $\boldsymbol{z}(t) = \big(\boldsymbol{x}(t), \boldsymbol{h}(t-1)\big)$

Update Equations

- Inputs $\boldsymbol{z}(t) = \big(\boldsymbol{x}(t), \boldsymbol{h}(t-1)\big)$
- Network updates ($\boldsymbol{W}_*$ are the free parameters)

$$\boldsymbol{f}(t) = \boldsymbol{\sigma}(\boldsymbol{W}_f\, \boldsymbol{z}(t)) \qquad \boldsymbol{i}(t) = \boldsymbol{\sigma}(\boldsymbol{W}_i\, \boldsymbol{z}(t))$$

$$\boldsymbol{g}(t) = \mathbf{tanh}(\boldsymbol{W}_g\, \boldsymbol{z}(t)) \qquad \boldsymbol{o}(t) = \boldsymbol{\sigma}(\boldsymbol{W}_o\, \boldsymbol{z}(t))$$

Update Equations

- Inputs $\boldsymbol{z}(t) = \big(\boldsymbol{x}(t), \boldsymbol{h}(t-1)\big)$
- Network updates ($\boldsymbol{W}_*$ are the free parameters)

$$\boldsymbol{f}(t) = \boldsymbol{\sigma}(\boldsymbol{W}_f\, \boldsymbol{z}(t)) \qquad \boldsymbol{i}(t) = \boldsymbol{\sigma}(\boldsymbol{W}_i\, \boldsymbol{z}(t))$$
$$\boldsymbol{g}(t) = \mathbf{tanh}(\boldsymbol{W}_g\, \boldsymbol{z}(t)) \qquad \boldsymbol{o}(t) = \boldsymbol{\sigma}(\boldsymbol{W}_o\, \boldsymbol{z}(t))$$

- Long-term memory update

$$\boldsymbol{c}(t) = \boldsymbol{f}(t) \otimes \boldsymbol{c}(t-1) + \boldsymbol{g}(t) \otimes \boldsymbol{i}(t)$$

Update Equations

- Inputs $\boldsymbol{z}(t) = \big(\boldsymbol{x}(t), \boldsymbol{h}(t-1)\big)$
- Network updates ($\boldsymbol{W}_*$ are the free parameters)

$$\boldsymbol{f}(t) = \boldsymbol{\sigma}(\boldsymbol{W}_f\, \boldsymbol{z}(t)) \qquad \boldsymbol{i}(t) = \boldsymbol{\sigma}(\boldsymbol{W}_i\, \boldsymbol{z}(t))$$
$$\boldsymbol{g}(t) = \mathbf{tanh}(\boldsymbol{W}_g\, \boldsymbol{z}(t)) \qquad \boldsymbol{o}(t) = \boldsymbol{\sigma}(\boldsymbol{W}_o\, \boldsymbol{z}(t))$$

- Long-term memory update

$$\boldsymbol{c}(t) = \boldsymbol{f}(t) \otimes \boldsymbol{c}(t-1) + \boldsymbol{g}(t) \otimes \boldsymbol{i}(t)$$

- Output $\boldsymbol{h}(t) = \boldsymbol{o}(t) \otimes \mathbf{tanh}(\boldsymbol{c}(t))$

Training LSTM

- We can train an LSTM by unwrapping it in time

Training LSTM

- We can train an LSTM by unwrapping it in time
- Note that it involves four dense layers with sigmoidal (or tanh) outputs

Training LSTM

- We can train an LSTM by unwrapping it in time
- Note that it involves four dense layers with sigmoidal (or tanh) outputs
- This means that typically it is very slow to train

Training LSTM

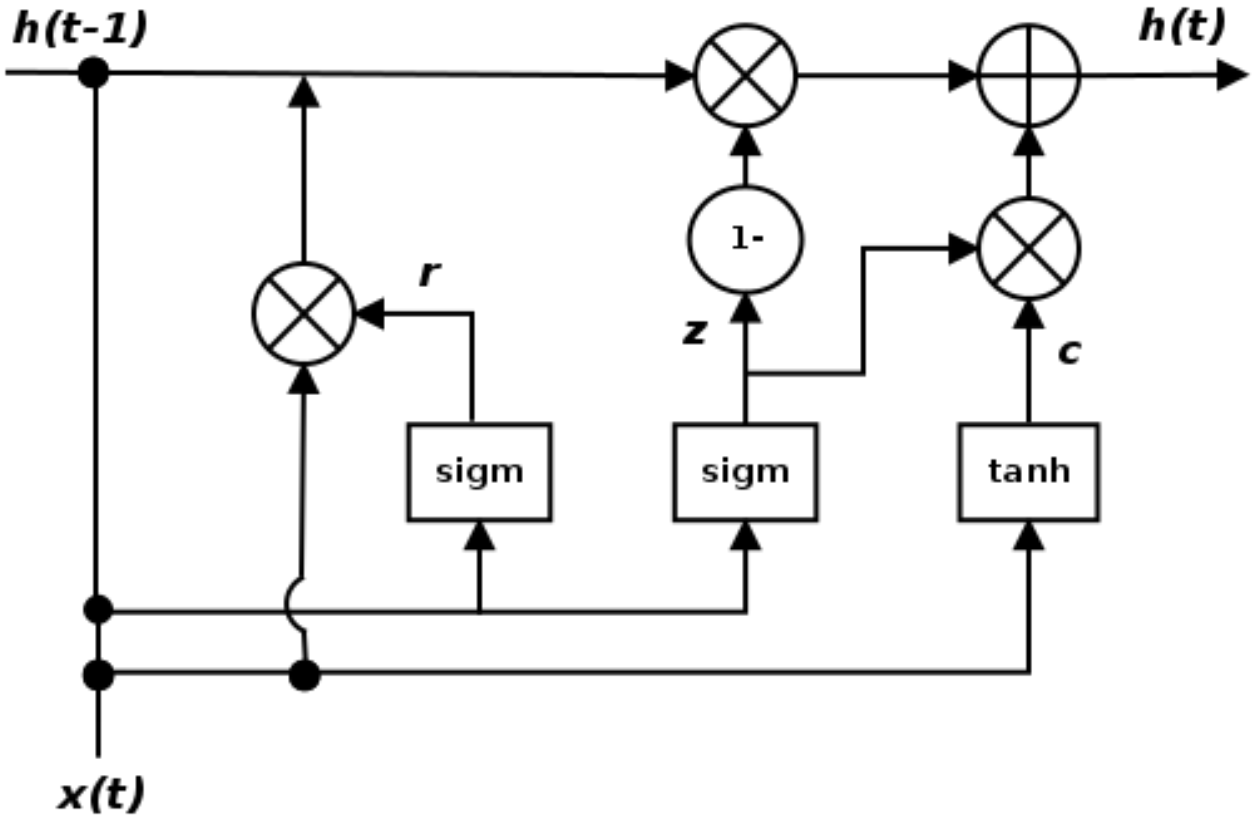
- We can train an LSTM by unwrapping it in time
- Note that it involves four dense layers with sigmoidal (or tanh) outputs
- This means that typically it is very slow to train
- There are a few variants of LSTMs, but all are very similar. The most popular is probably Gated Recurrent Unit (GRU)

LSTM Success Stories

- LSTMs have been used to win many competitions in speech and handwriting recognition
- Major technology companies including Google, Apple, and Microsoft are using LSTMs as fundamental components in new products.
- Google used LSTM for speech recognition on the smartphone, for Google Translate.
- Apple uses LSTM for the "Quicktype" function on the iPhone and for Siri.
- Amazon uses LSTM for Amazon Alexa.
- In 2017, Facebook performed some 4.5 billion automatic translations every day using long short-term memory networks [1]

[1] https://en.wikipedia.org/wiki/Long_short-term_memory

Gated Recurrent Unit (GRU)



Gated Recurrent Unit (GRU)

- x_t: input vector
- h_t: output vector
- z_t: update gate vector
- r_t: reset gate vector
- W, U, and b: parameter matrices and vector
- *sigm* or σ_g is the sigmoid function
- *tanh* or σ_h is the hyperbolic tangent

Gated Recurrent Unit (GRU)

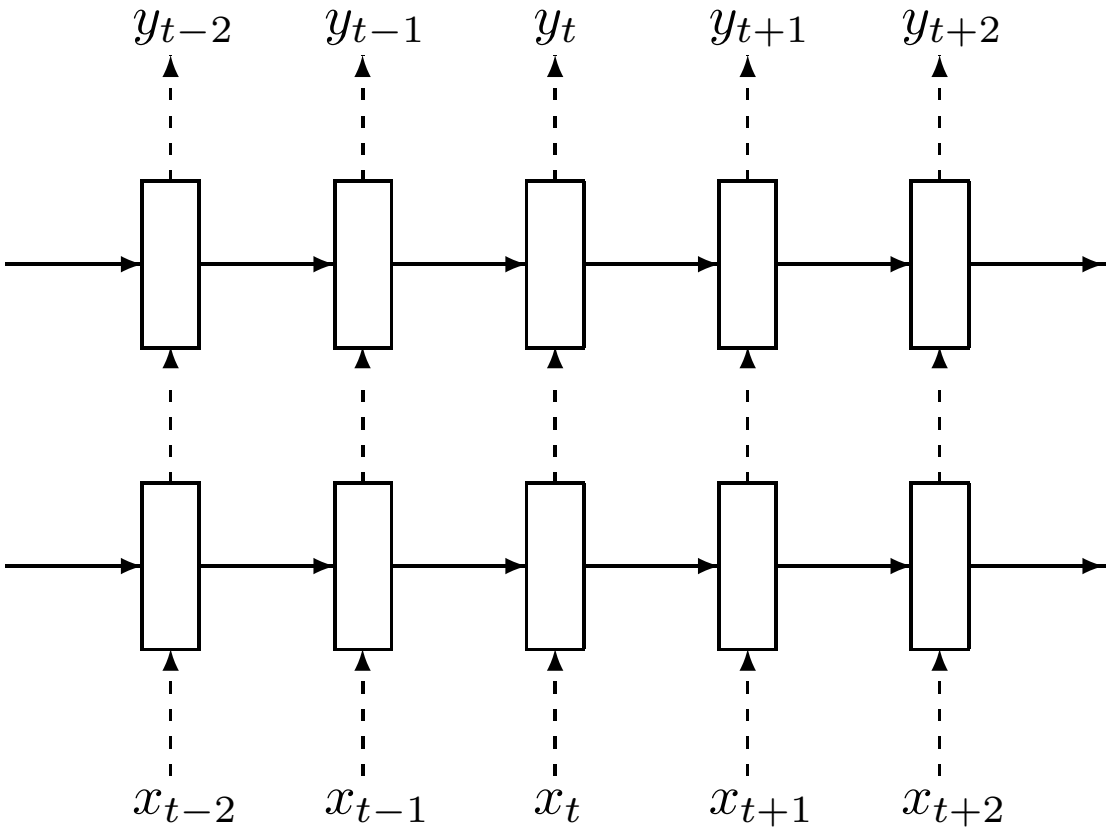
Initially, for $t = 0$, $h_0 = 0$

$$
\begin{aligned}
z_t &= \sigma_g(W_z x_t + U_z h_{t-1} + b_z) \\
r_t &= \sigma_g(W_r x_t + U_r h_{t-1} + b_r) \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \sigma_h(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)
\end{aligned}
$$

GRU vs. LSTM?

- GRUs have two gates (reset and update) whereas LSTM has three gates (input/output/forget)
- GRU performance on par with LSTM but computationally more efficient (less complex).
- In general, if you have a very large dataset then LSTMs will likely perform better.
- GRUs are a good choice for smaller datasets.

Regularization in RNNs with LSTM units



2

[2] https://arxiv.org/pdf/1409.2329.pdf

Regularization in RNNs with LSTM units

$$\text{LSTM} : h_t^{l-1}, h_{t-1}^l, c_{t-1}^l \rightarrow h_t^l, c_t^l$$

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} T_{2n,4n} \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

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

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

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} T_{2n,4n} \begin{pmatrix} \mathbf{D}(h_t^{l-1}) \\ h_{t-1}^l \end{pmatrix}$$

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

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

3

[3] https://arxiv.org/pdf/1409.2329.pdf