# Forget to remember
# Remember to forget

VLC = ∭ Vision Learning Control dVdLdC

# Long Short Term Memories and Gated Recurrent Units

Jonathon Hare

Vision, Learning and Control
University of Southampton

Some of the images and animations used here were originally designed 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 $a^{\tau-1}$

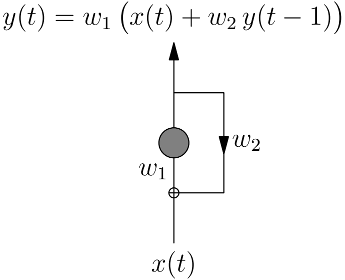
# Recap: An RNN is just a recursive function invocation

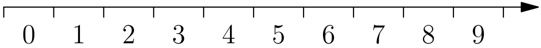
- $\boldsymbol{y}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{c}(t-1)|\boldsymbol{W})$
- and the state $\boldsymbol{c}(t) = \boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{c}(t-1)|\boldsymbol{W})$
- If the output $\boldsymbol{y}(t)$ depends on the input $\boldsymbol{x}(t-2)$, then prediction will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

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

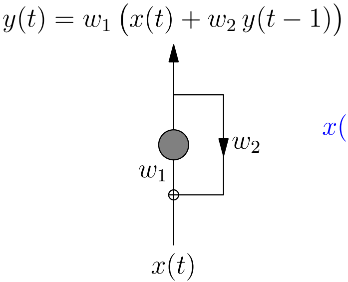
- it should be clear that the gradients of this with respect to the weights can be found with the chain rule
- The back-propagated error will involve applying $\boldsymbol{f}$ multiple times
- Each time the error will get multiplied by some factor $a$
- If $\boldsymbol{y}(t)$ depends on the input $\boldsymbol{x}(t-\tau)$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

# Vanishing and Exploding Gradients

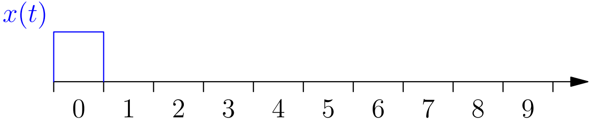


# Vanishing and Exploding Gradients

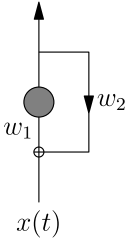



$$w_1 = w_2 = 0.9$$

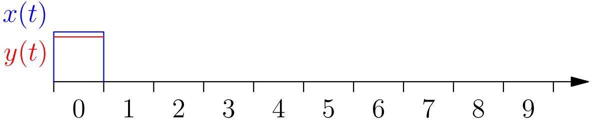


# Vanishing and Exploding Gradients

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

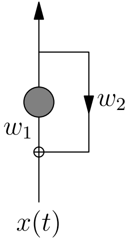



$$w_1 = w_2 = 0.9$$

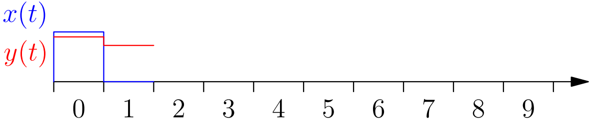


# Vanishing and Exploding Gradients

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

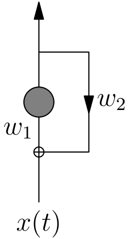



$$w_1 = w_2 = 0.9$$

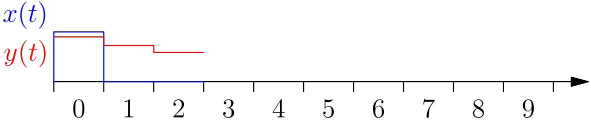


# Vanishing and Exploding Gradients

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

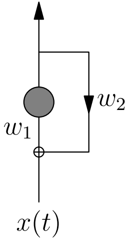



$$w_1 = w_2 = 0.9$$

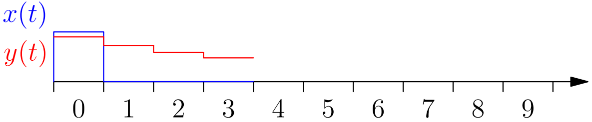


# Vanishing and Exploding Gradients

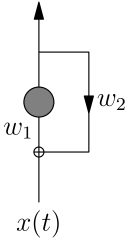
$$y(t) = w_1 \left(x(t) + w_2\, y(t-1)\right)$$



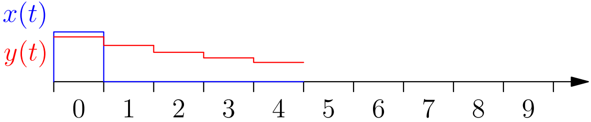

$$w_1 = w_2 = 0.9$$



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

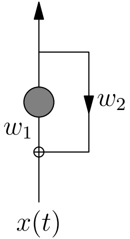



$$w_1 = w_2 = 0.9$$

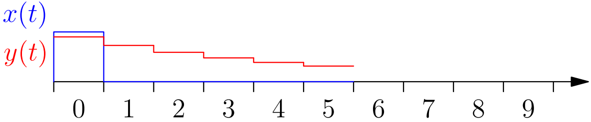


# Vanishing and Exploding Gradients

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

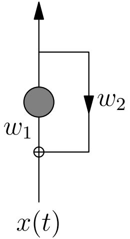



$$w_1 = w_2 = 0.9$$

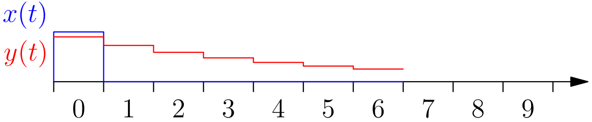


# Vanishing and Exploding Gradients

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



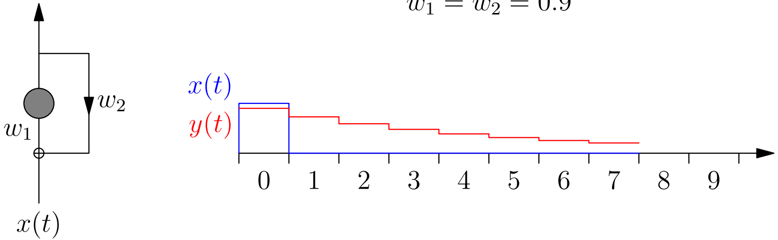

$$w_1 = w_2 = 0.9$$



# Vanishing and Exploding Gradients

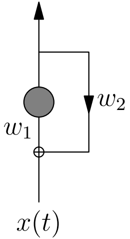
$$y(t) = w_1 \left(x(t) + w_2\, y(t-1)\right)$$

$$w_1 = w_2 = 0.9$$

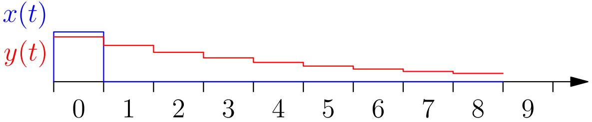


# Vanishing and Exploding Gradients

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

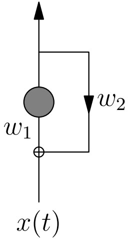



$$w_1 = w_2 = 0.9$$

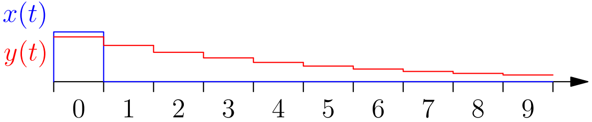


# Vanishing and Exploding Gradients

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

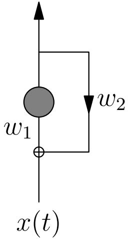



$$w_1 = w_2 = 0.9$$

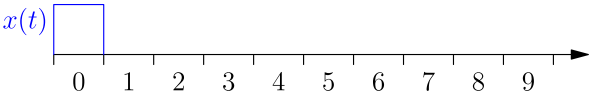


# Vanishing and Exploding Gradients

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

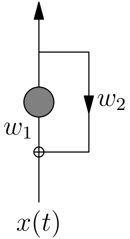



$$w_1 = w_2 = 1.1$$

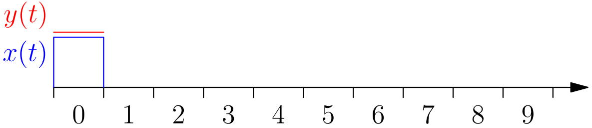


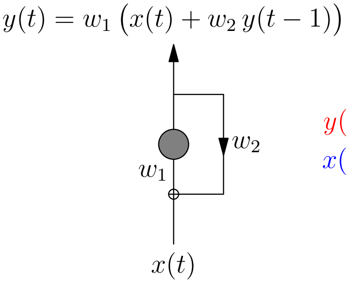
# Vanishing and Exploding Gradients

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

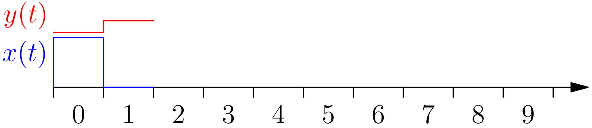



$$w_1 = w_2 = 1.1$$

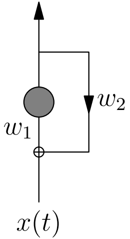




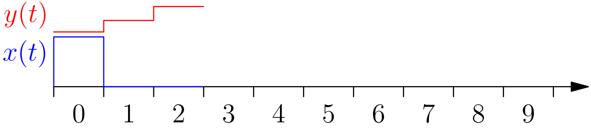

$w_1 = w_2 = 1.1$



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

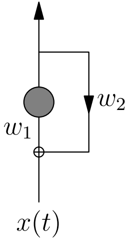



$$w_1 = w_2 = 1.1$$

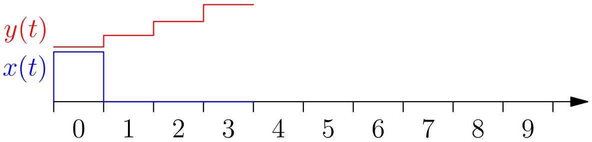


# Vanishing and Exploding Gradients

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

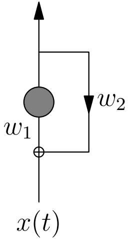



$$w_1 = w_2 = 1.1$$

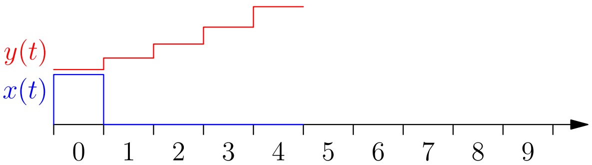


# Vanishing and Exploding Gradients

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

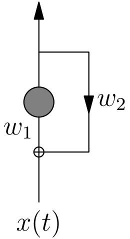


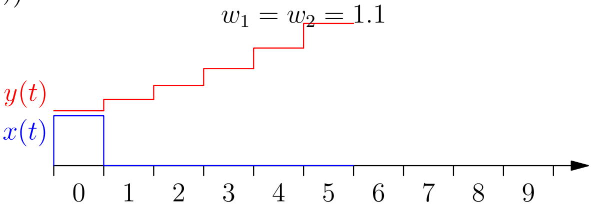

$$w_1 = w_2 = 1.1$$



# Vanishing and Exploding Gradients

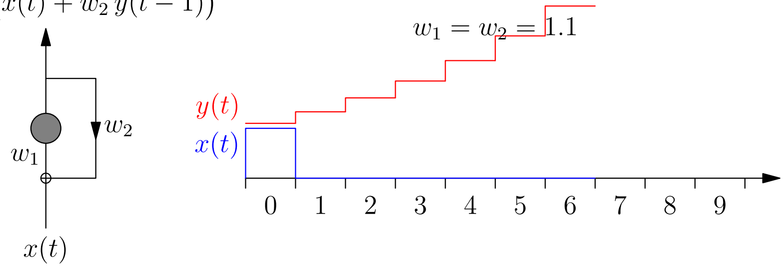
$$y(t) = w_1 \left(x(t) + w_2\, y(t-1)\right)$$



# Vanishing and Exploding Gradients

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



# 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

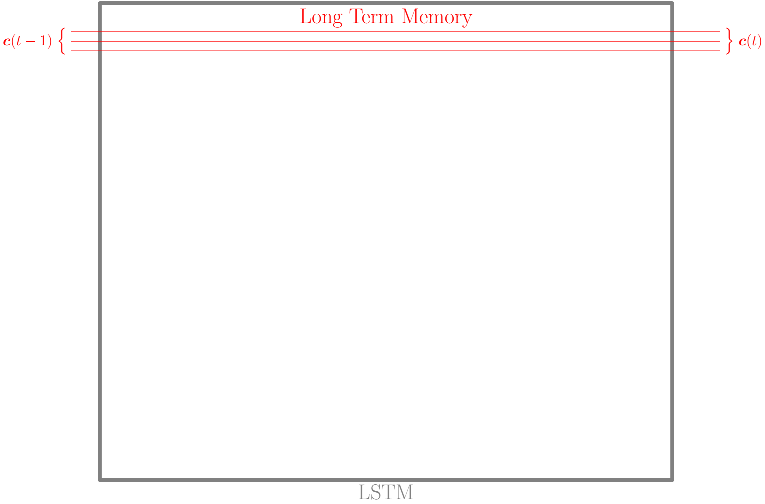
$$\boldsymbol{c}(t) = \boldsymbol{c}(t-1)$$

- 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

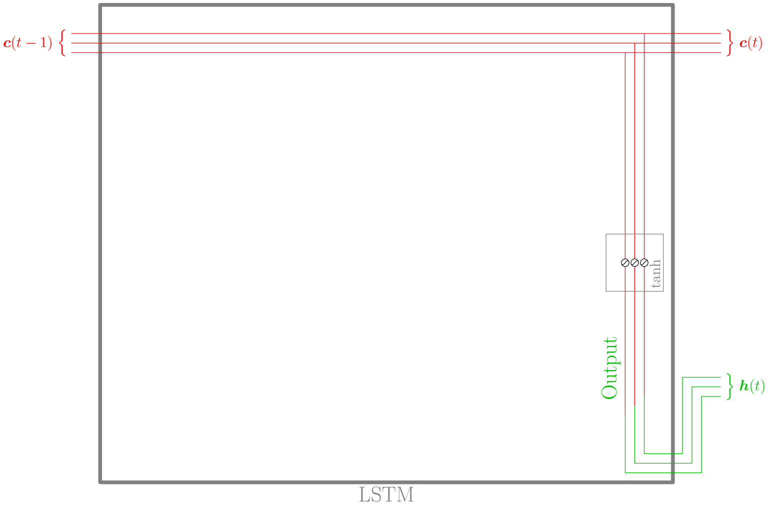
# LSTM Architecture

LSTM

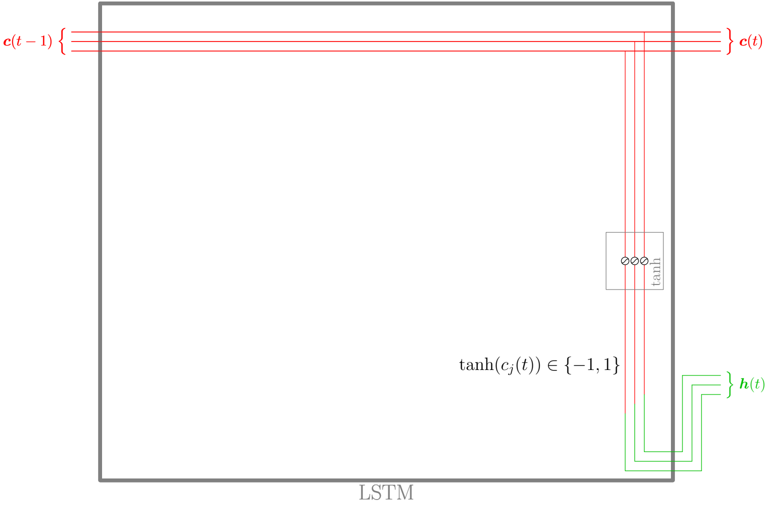
# LSTM Architecture



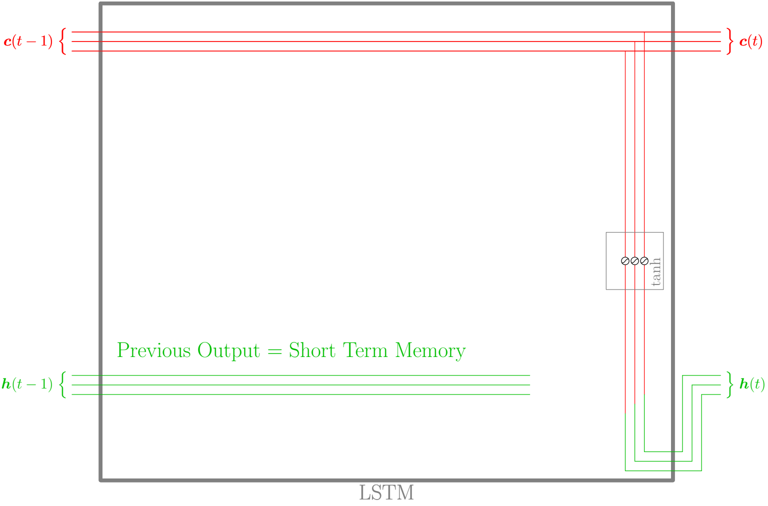
# LSTM Architecture



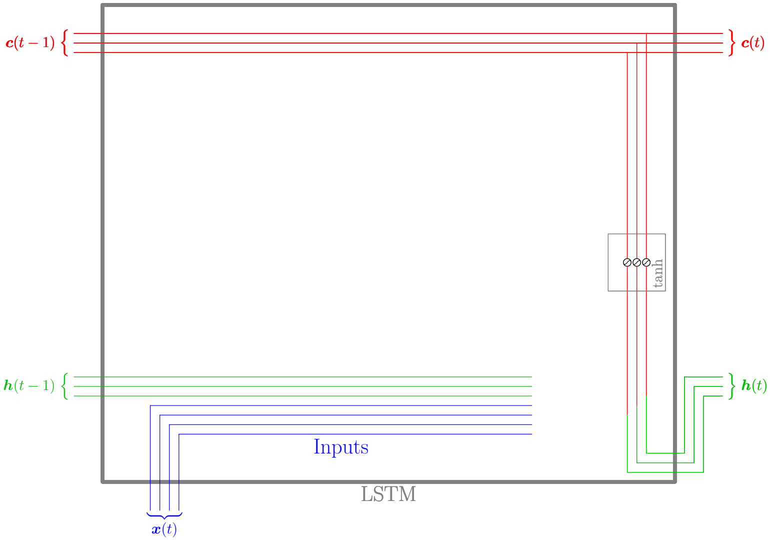
# LSTM Architecture



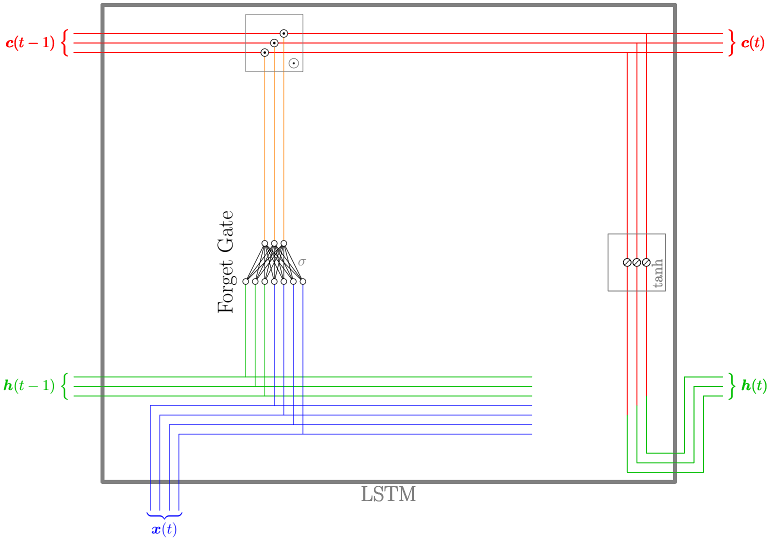
# LSTM Architecture



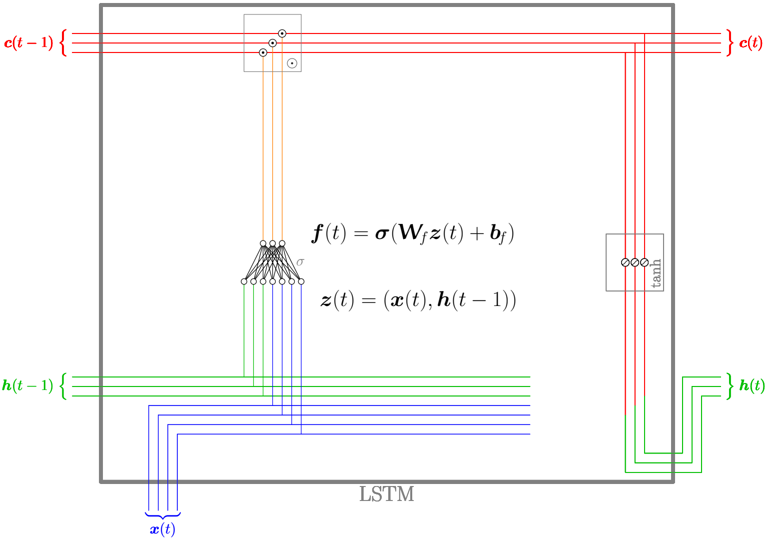
# LSTM Architecture



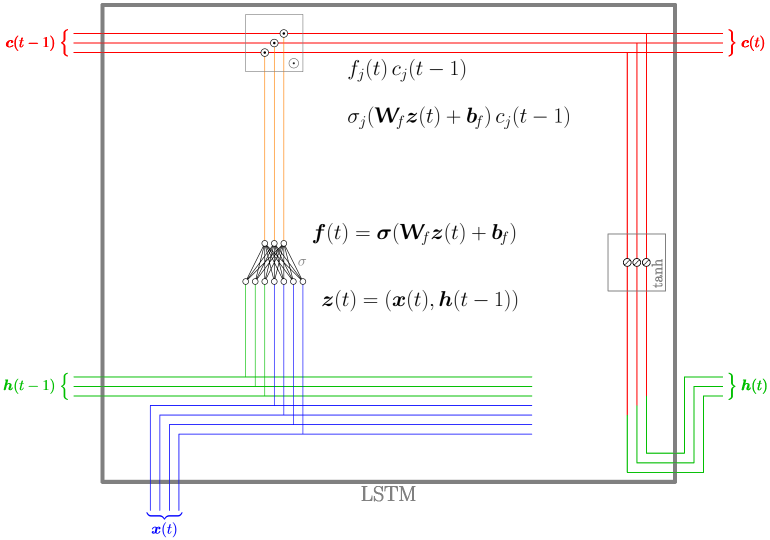
# LSTM Architecture



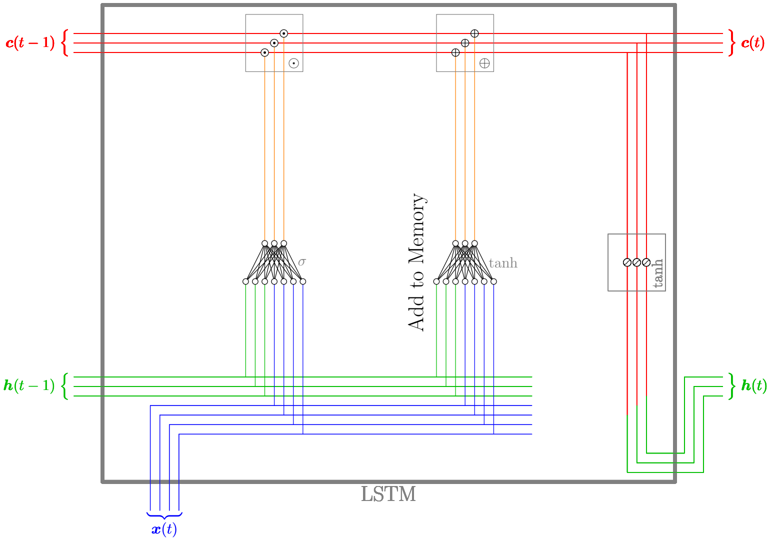
# LSTM Architecture



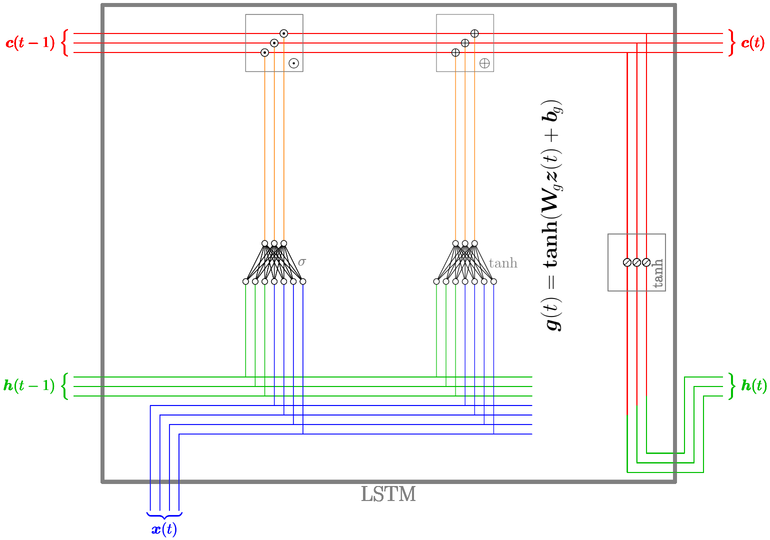
# LSTM Architecture



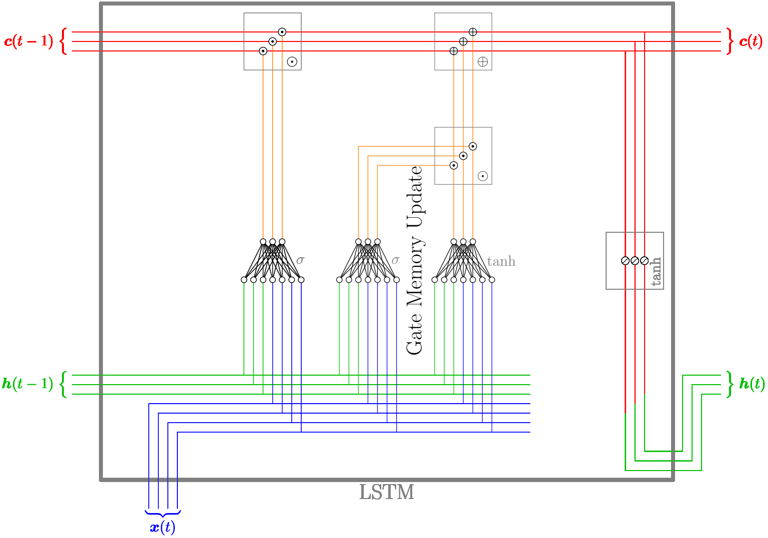
# LSTM Architecture



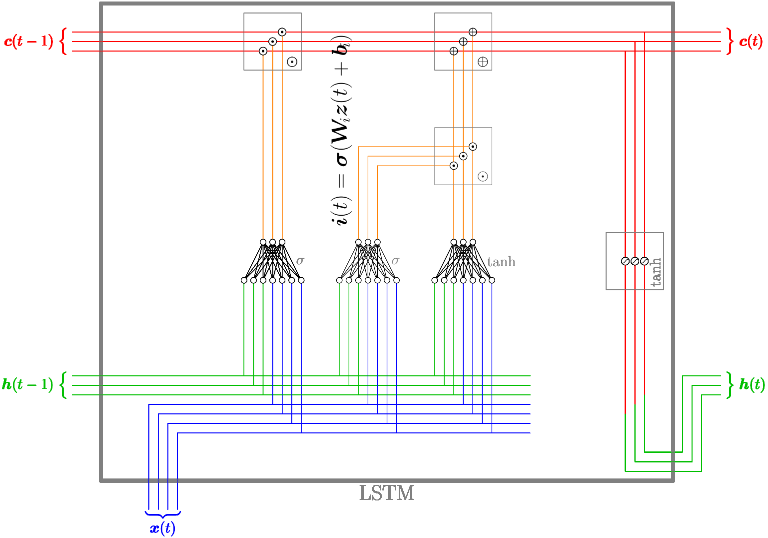
# LSTM Architecture



# LSTM Architecture

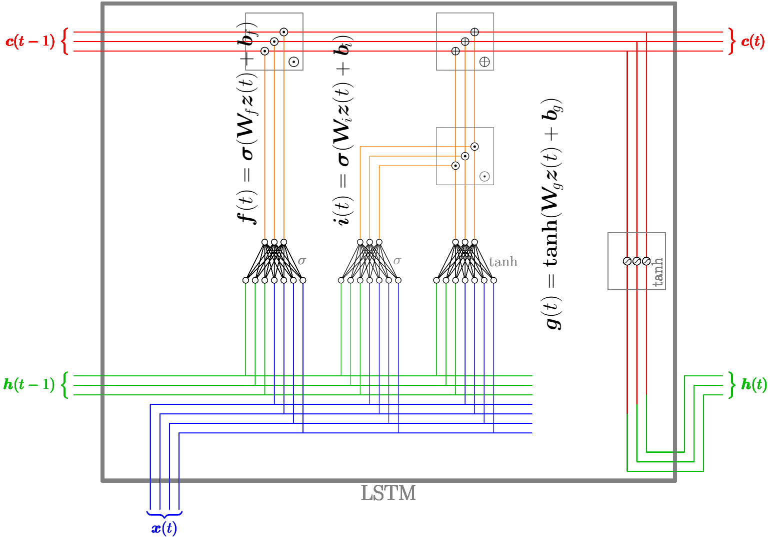


# LSTM Architecture

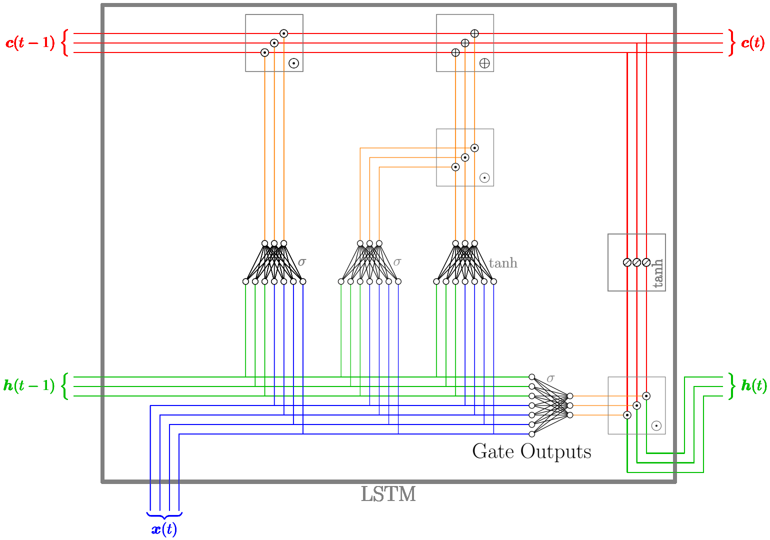


# LSTM Architecture

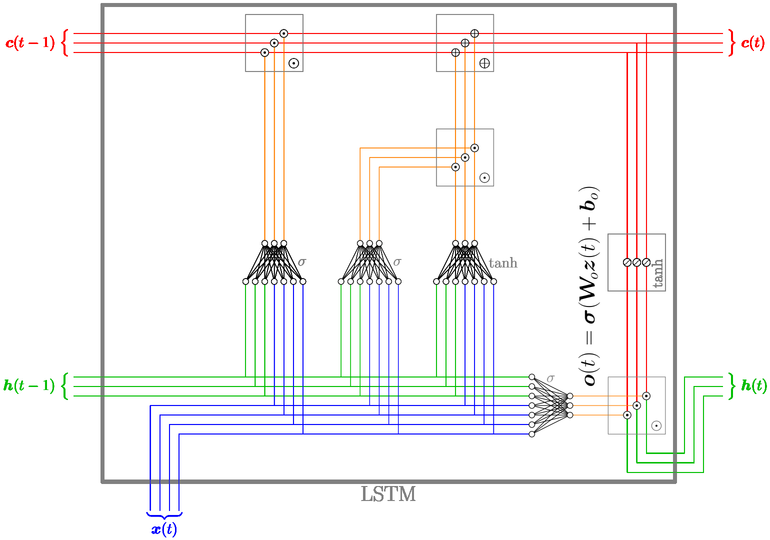
$$c_j(t) = f_j(t)\, c_j(t-1) + b_f + g_j(t)\, i_j(t) + b_i$$



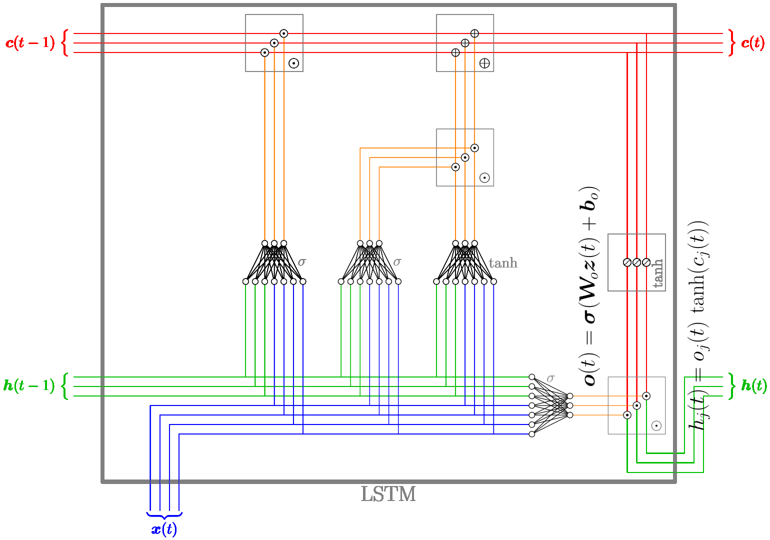
# LSTM Architecture



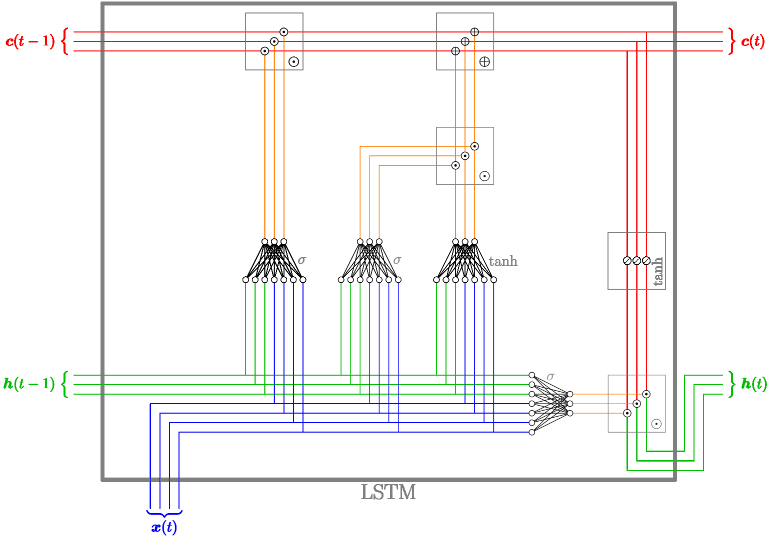
# LSTM Architecture



# LSTM Architecture



# LSTM Architecture



# Update Equations

Initially, for $t = 0$, $\boldsymbol{h}(0) = \mathbf{0}$

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

# Update Equations

Initially, for $t = 0$, $\boldsymbol{h}(0) = \mathbf{0}$

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

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

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

# Update Equations

Initially, for $t = 0$, $\boldsymbol{h}(0) = \mathbf{0}$

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

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

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

- Long-term memory update

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

# Update Equations

Initially, for $t = 0$, $\boldsymbol{h}(0) = \mathbf{0}$

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

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

- Long-term memory update

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

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

# Training LSTMs

- We can train an LSTM by unwrapping it in time.

# Training LSTMs

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

# Training LSTMs

- 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.

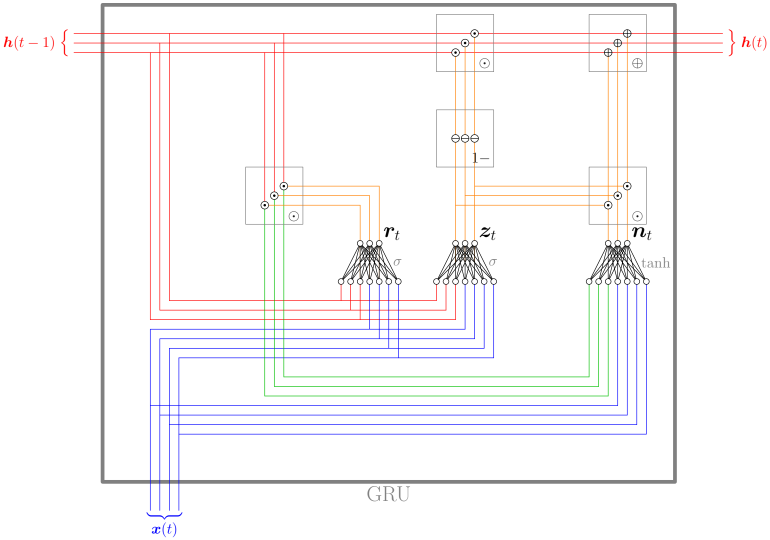
# Training LSTMs

- 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 the 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 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


$\boldsymbol{h}(t-1)$
$\boldsymbol{h}(t)$
1−
$\boldsymbol{r}_t$
$\boldsymbol{z}_t$
$\boldsymbol{n}_t$
$\sigma$
$\sigma$
tanh
GRU
$\boldsymbol{x}(t)$

# Gated Recurrent Unit (GRU)

- $\boldsymbol{x}(t)$: input vector
- $\boldsymbol{h}(t)$: output vector (and 'hidden state')
- $\boldsymbol{r}(t)$: reset gate vector
- $\boldsymbol{z}(t)$: update gate vector
- $\boldsymbol{n}(t)$: new state vector (before update is applied)
- $\boldsymbol{W}$ and $\boldsymbol{b}$: parameter matrices and biases

# Gated Recurrent Unit (GRU)

Initially, for $t = 0$, $\boldsymbol{h}(0) = \mathbf{0}$

$$\begin{aligned}
\boldsymbol{z}(t) &= \sigma(\boldsymbol{W}_z(\boldsymbol{x}(t), \boldsymbol{h}(t-1)) + \boldsymbol{b}_z) \\
\boldsymbol{r}(t) &= \sigma(\boldsymbol{W}_r(\boldsymbol{x}(t), \boldsymbol{h}(t-1)) + \boldsymbol{b}_r) \\
\boldsymbol{n}(t) &= \tanh(\boldsymbol{W}_n(\boldsymbol{x}(t), r(t) \odot h(t-1)) + \boldsymbol{b}_h) \\
\boldsymbol{h}(t) &= (1 - \boldsymbol{z}(t)) \odot \boldsymbol{h}(t-1) + \boldsymbol{z}(t) \odot \boldsymbol{n}(t)
\end{aligned}$$

Most implementations follow the original paper and swap $(1 - \boldsymbol{z}(t))$ and $(\boldsymbol{z}(t))$ in the $\boldsymbol{h}(t)$ update; this doesn't change the operation of the network, but does change the interpretation of the update gate, as the gate would have to produce a 0 when an update was to occur, and a 1 when no update is to happen (which is somewhat counter-intuitive)!

# GRU or 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 operations & weights).
- In general, if you have a very large dataset then LSTMs will likely perform slightly better.
- GRUs are a good choice for smaller datasets.