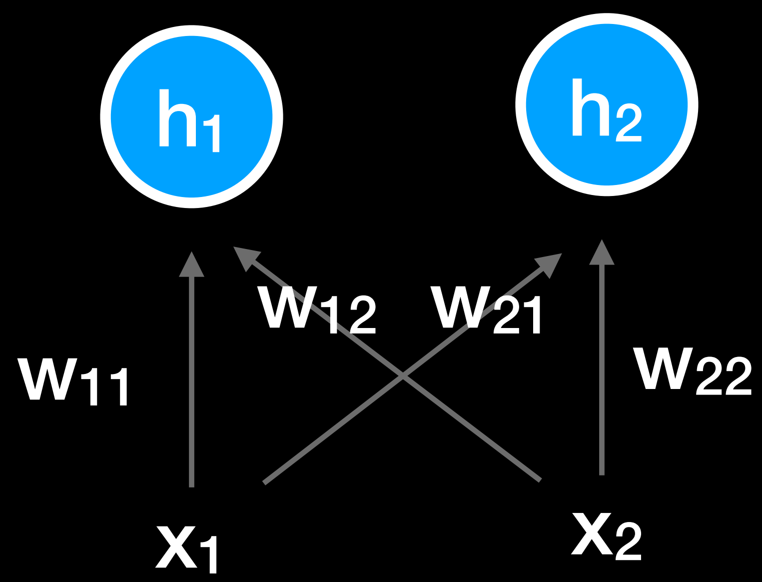
Recurrent neural networks

Nicholas Beshouri — nbeshouri@gmail.com

Neural nets review



$$\mathbf{h} = \text{foo}(\mathbf{Wx} + \mathbf{b})$$

$$\boxed{\begin{matrix} \text{foo}(w_{11}x_1 + w_{12}x_2 + b_1) \\ \text{foo}(w_{21}x_1 + w_{22}x_2 + b_2) \end{matrix}} = \text{foo}\left[\boxed{\begin{matrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{matrix}} \boxed{\begin{matrix} x_1 \\ x_2 \end{matrix}} + \boxed{\begin{matrix} b_1 \\ b_2 \end{matrix}} \right]$$

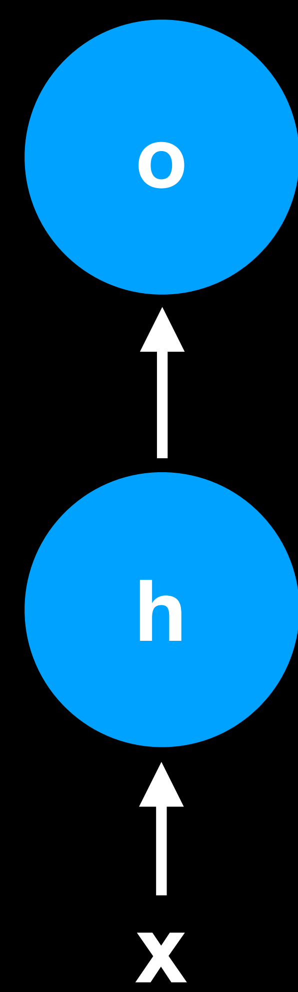
Why do we need RNNs?

Motivating example

Clouds are in the ______

Motivating example

You might be tempted to try a standard dense network...

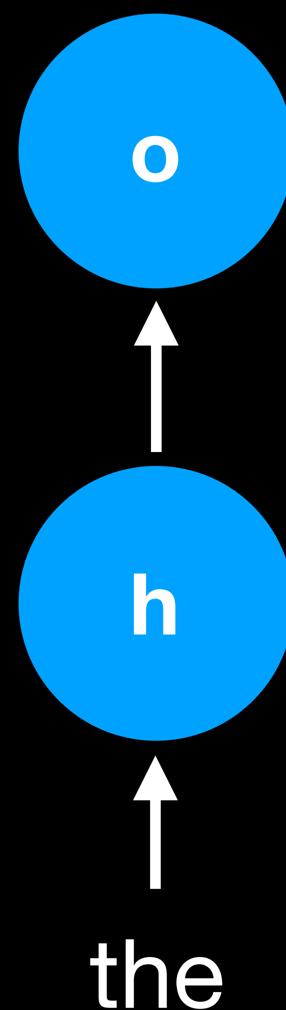


$$\mathbf{o} = \mathrm{softmax}(\mathbf{W}_o\mathbf{h} + \mathbf{b}_o)$$

$$\mathbf{h} = \mathrm{relu}(\mathbf{W}_h\mathbf{x} + \mathbf{b}_h)$$

Motivating example

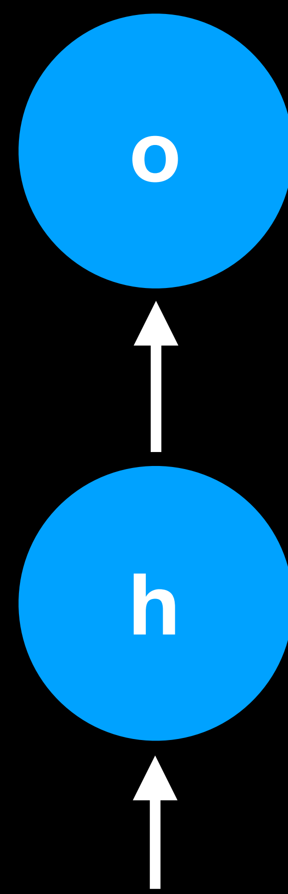
But just looking at the last word isn't enough.



$$\mathbf{o} = \mathrm{softmax}(\mathbf{W}_o\mathbf{h} + \mathbf{b}_o)$$

$$\mathbf{h} = \mathrm{relu}(\mathbf{W}_h\mathbf{x} + \mathbf{b}_h)$$

Motivating example

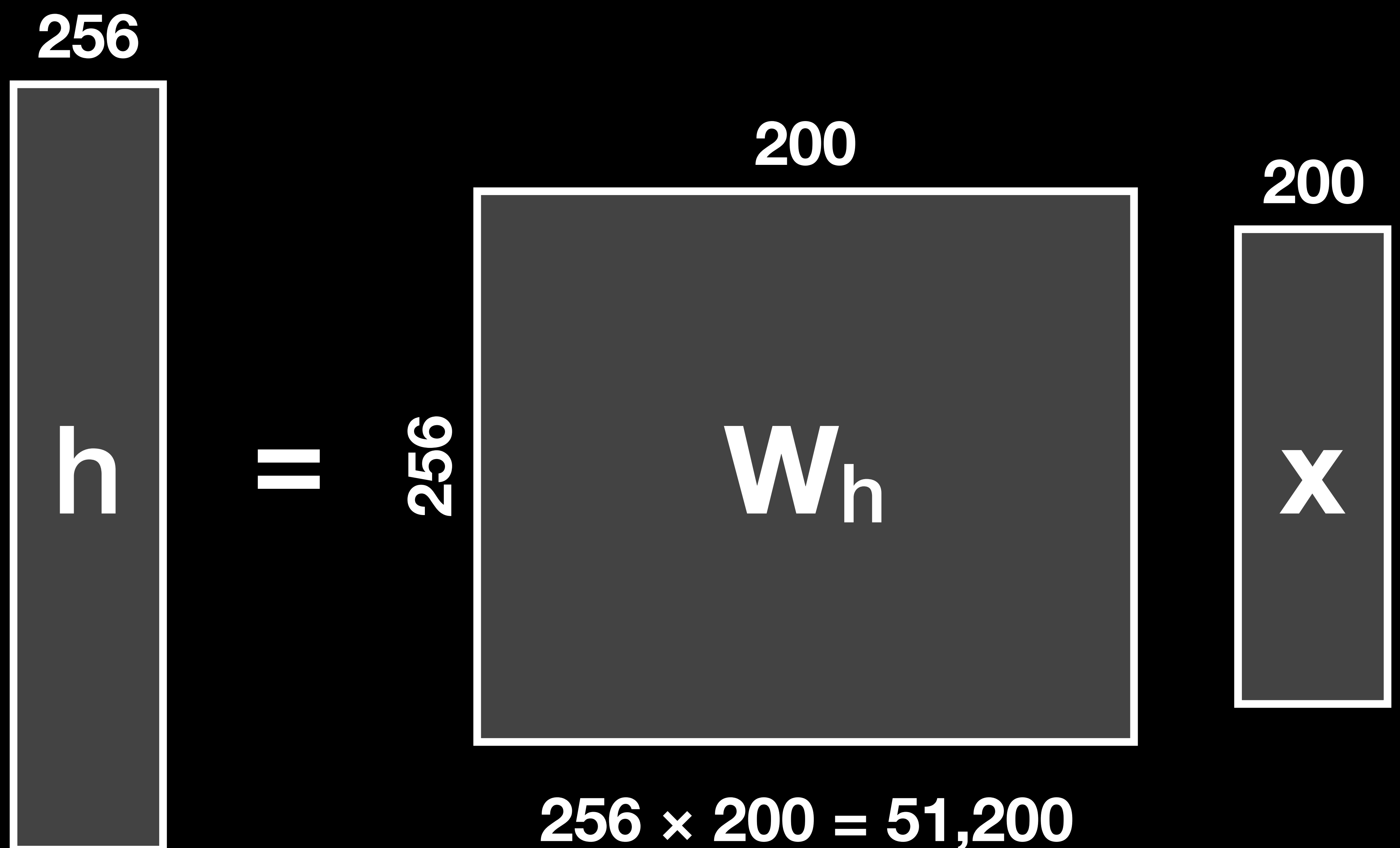


$$\mathbf{o} = \mathrm{softmax}(\mathbf{W}_o\mathbf{h} + \mathbf{b}_o)$$

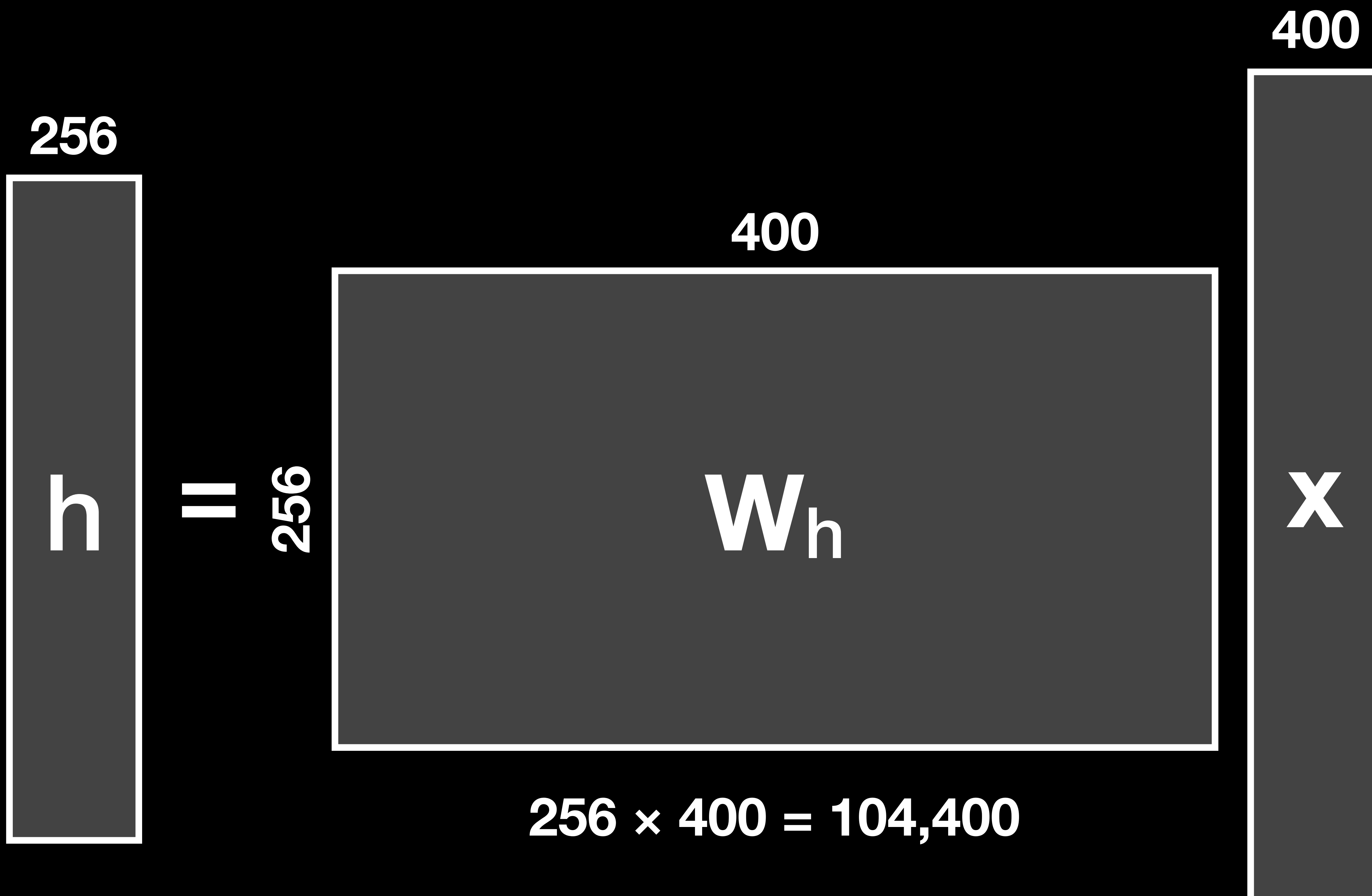
$$\mathbf{h} = \mathrm{relu}(\mathbf{W}_h\mathbf{x} + \mathbf{b}_h)$$

[in, the]

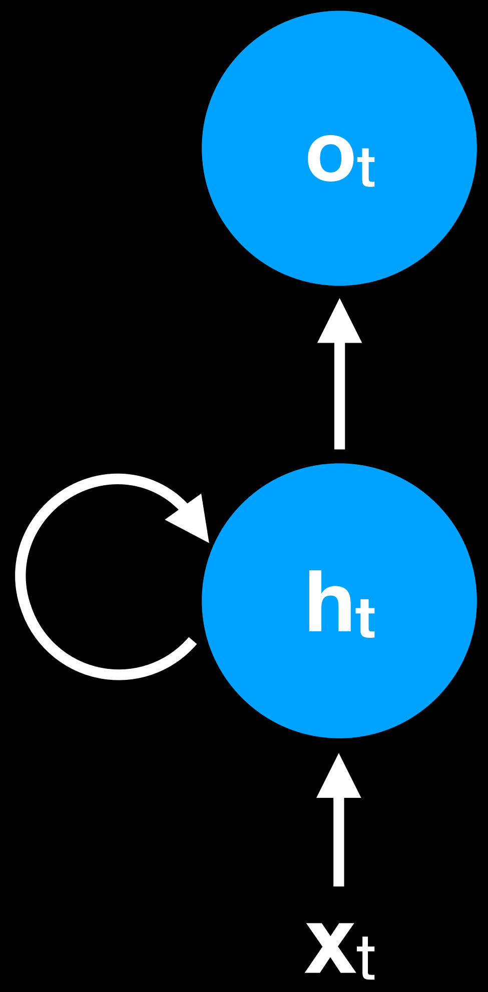
Motivating example



Motivating example



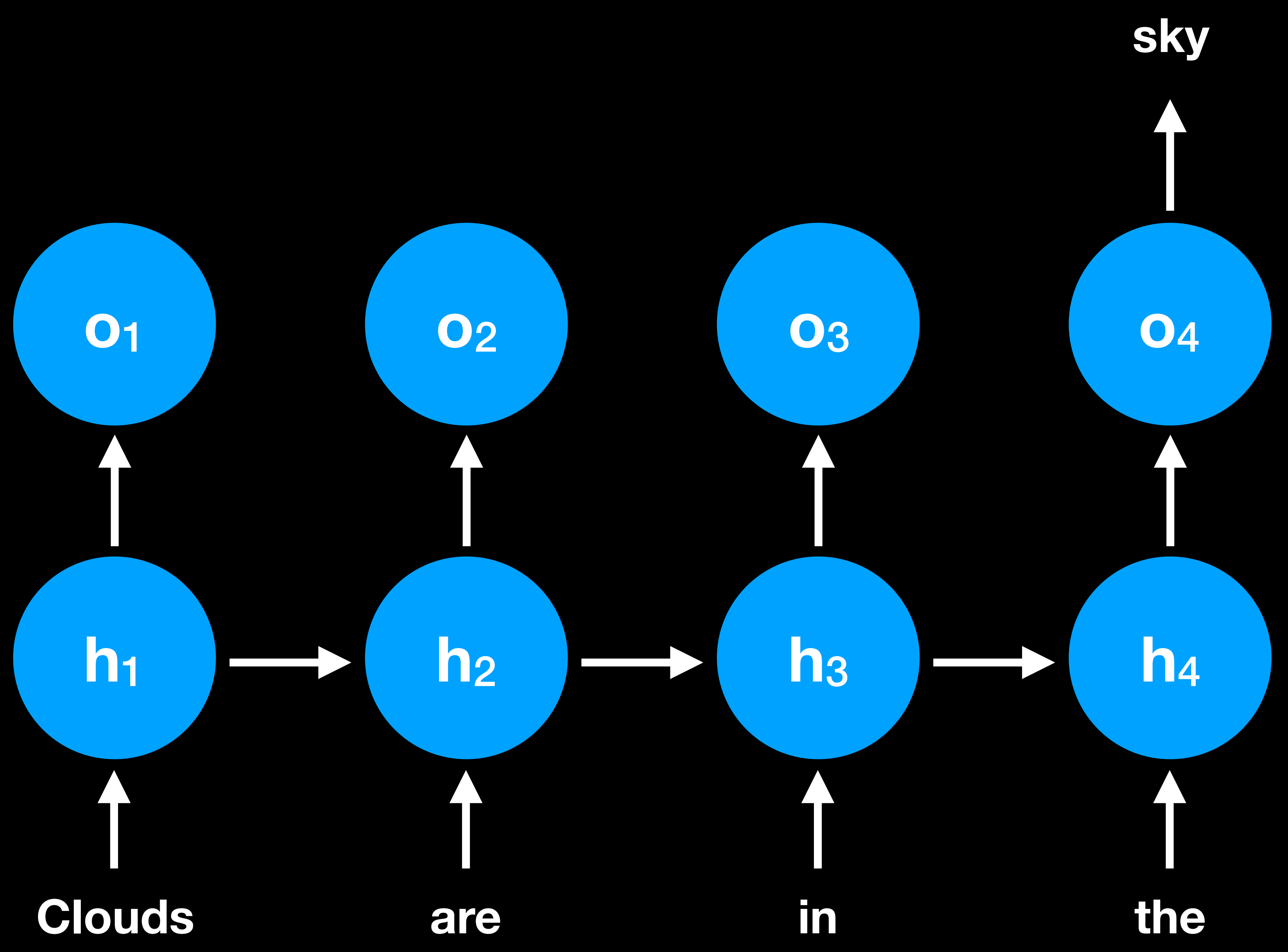
Vanilla RNNs



$$\mathbf{o}_t = \mathrm{softmax}(\mathbf{W}_o\mathbf{h}_t + \mathbf{b}_h)$$

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

Vanilla RNNs



But they don't work...

The problems with vanilla RNNs

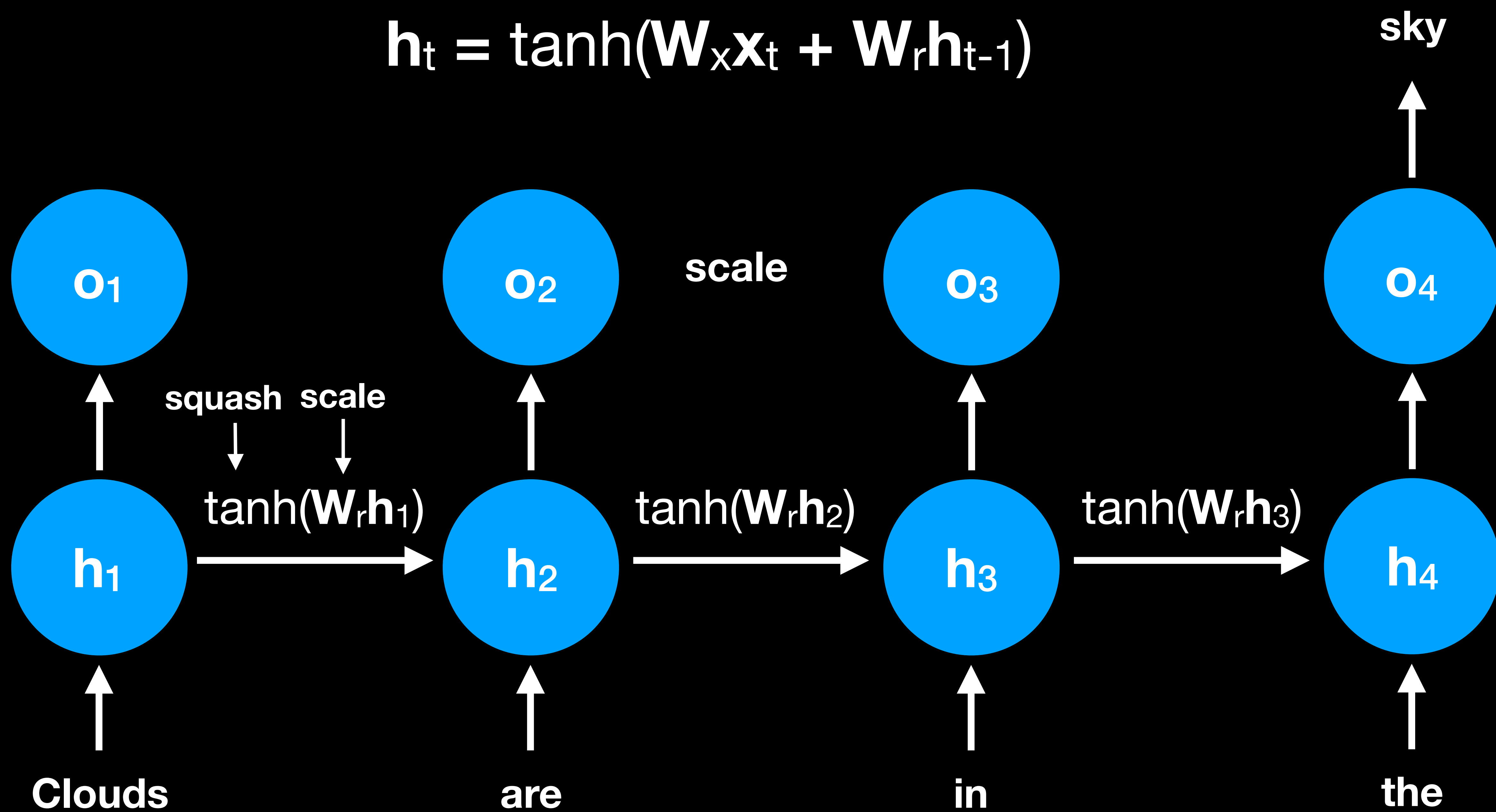
1. Problems going forwards: Signals from earlier in time become fainter and fainter.

2. Problems going backwards: Gradients vanish or explode.

Going forwards

Going forwards

$$\mathbf{h}_t = \tanh(\mathbf{W}_x\mathbf{x}_t + \mathbf{W}_r\mathbf{h}_{t-1})$$



Going forwards

```
In [22]:  1  Wr = np.random.normal(0, 3, (2, 2))
          2  Wr

Out[22]: array([[-1.15011703, -0.45383375],
                [-3.68550987,  2.13596934]])
```

Going forwards

```
In [22]: Wr = np.random.normal(0, 3, (2, 2))
         Wr

Out[22]: array([[-1.15011703, -0.45383375],
                [-3.68550987,  2.13596934]])

In [25]: h = np.array([0.3, 0.2])
         for i in range(10000):
             h = Wr @ h
             h /= np.linalg.norm(h)
         h

Out[25]: array([ 0.12065226, -0.99269483])
```

scale

*squash

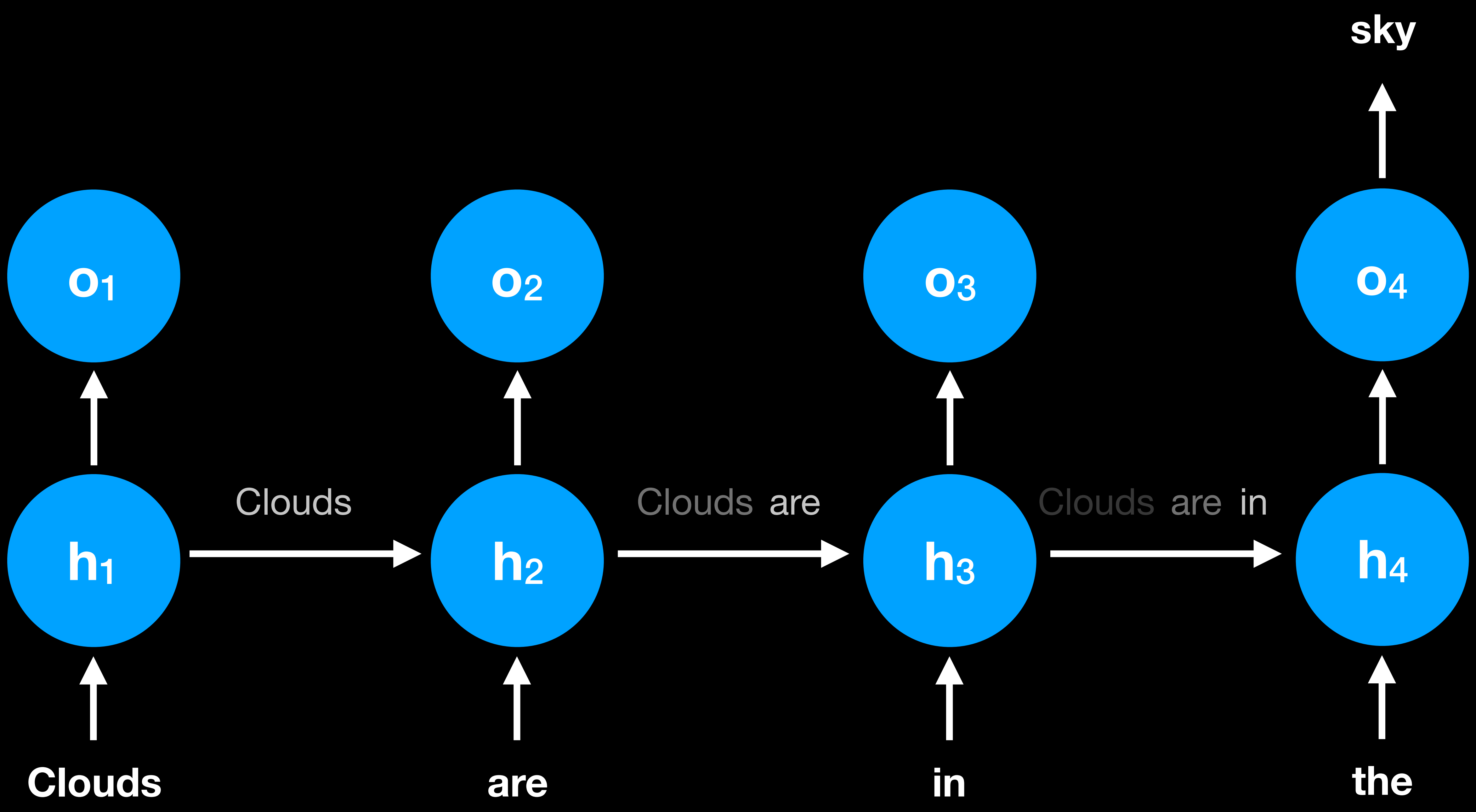
*This is a different kind of squashing than in the RNN, but the same idea applies: repeatedly scaling a vector unevenly then squashing it results in signal loss.

Going forwards

```
In [22]: Wr = np.random.normal(0, 3, (2, 2))
         Wr

Out[22]: array([[-1.15011703, -0.45383375],
                [-3.68550987,  2.13596934]])

In [25]: h = np.array([0.3, 0.2])
         for i in range(10000):
             h = Wr @ h
             h /= np.linalg.norm(h)
         h

Out[25]: array([ 0.12065226, -0.99269483])

In [23]: np.linalg.eig(Wr)[1]

Out[23]: array([[-0.7117151 ,  0.12065226],
                [-0.70246823, -0.99269483]])

In [24]: np.linalg.eig(Wr)[0]

Out[24]: array([-1.5980544 ,  2.58390671])
```

We'd get this output regardless of the original value of h. It's the eigenvector with the largest eigenvalue.

Going forwards



Going backwards

Backprop review

$$\frac{\partial Mx}{\partial x} = ?$$

Backprop review

$$\frac{\partial \mathbf{Mx}}{\partial \mathbf{x}} = \boxed{\begin{matrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} \end{matrix}}$$

Backprop review

$$\begin{bmatrix} m_{11}x_1 & m_{12}x_2 \\ m_{21}x_1 & m_{22}x_2 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix}$$

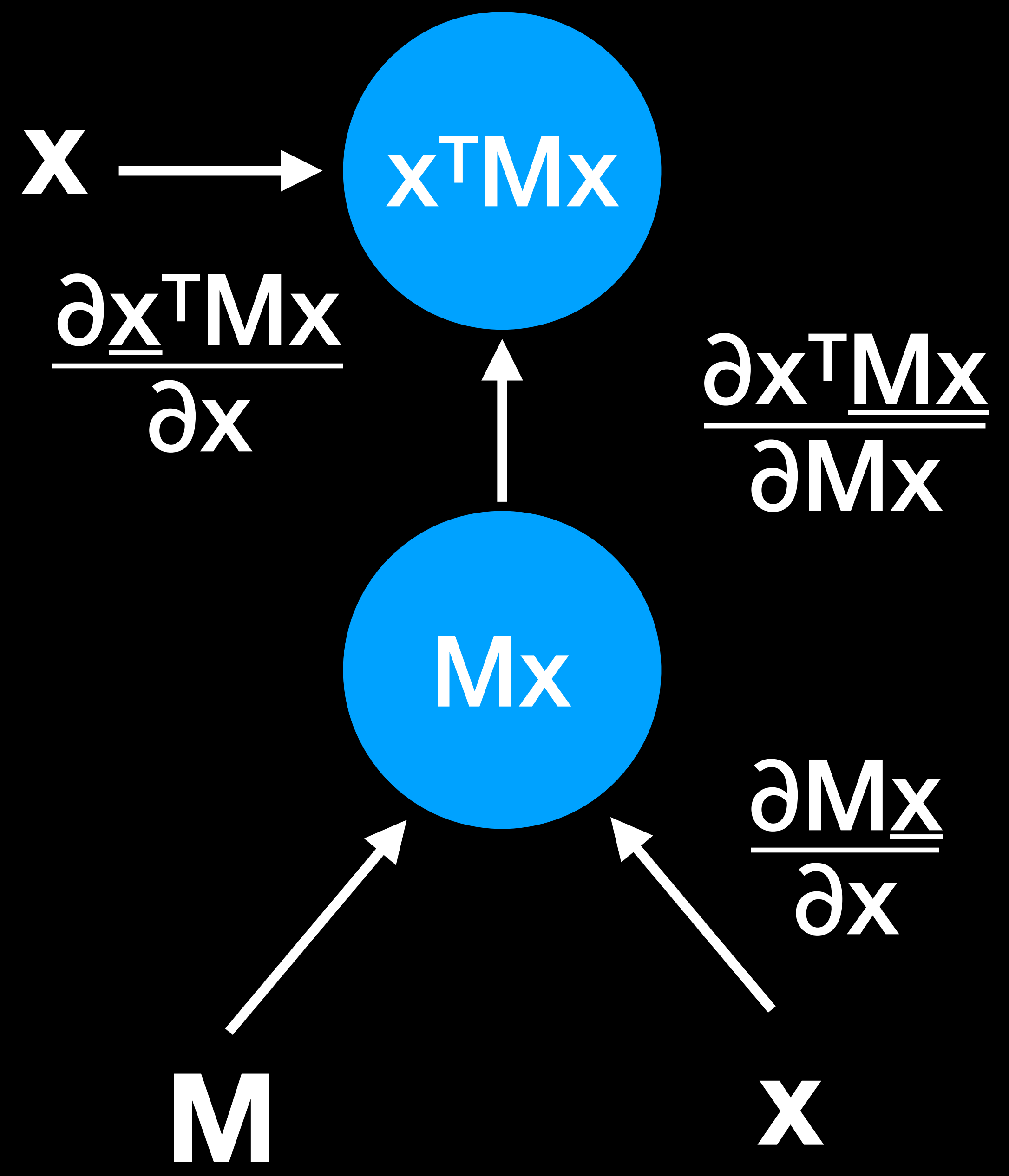
Backprop review

$$\frac{\partial \mathbf{M}\mathbf{x}}{\partial \mathbf{x}} = \mathbf{M}$$

Backprop review

$$\frac{\partial \mathbf{x}^T \mathbf{M} \mathbf{x}}{\partial \mathbf{x}} = ?$$

Backprop review



Backprop review

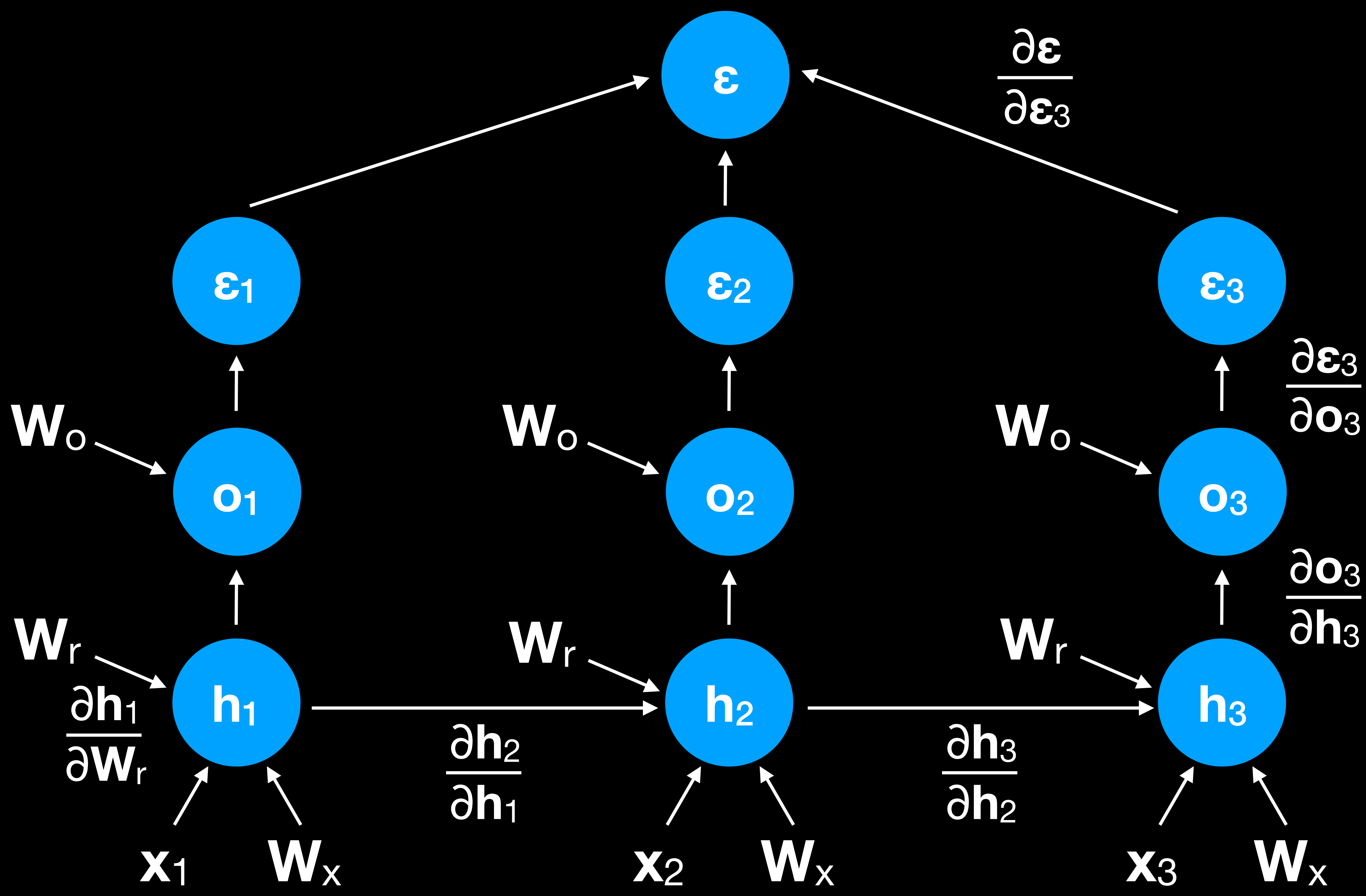
$$(\underline{x}^T Mx)^T = x^T M^T \underline{x}$$

$$\frac{\partial x^T Mx}{\partial x} = \frac{\partial \underline{x}^T Mx}{\partial x} + \frac{\partial x^T \underline{\underline{Mx}}}{\partial Mx} \frac{\partial M\underline{\underline{x}}}{\partial x}$$

$$= \quad x^T M^T \quad + \quad x^T \quad\quad M$$

$$= x^T M^T + x^T M$$

RNN gradients



RNN gradients

$$\frac{\partial \boldsymbol{\varepsilon}_3}{\partial \mathbf{o}_3} = \frac{\partial \boldsymbol{\varepsilon}_3}{\partial \mathbf{o}_3} \frac{\partial \mathbf{o}_3}{\partial \mathbf{h}_3} \boxed{\frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2} \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1}} \frac{\partial \mathbf{h}_1}{\partial \mathbf{W}_r} + \frac{\partial \boldsymbol{\varepsilon}_3}{\partial \mathbf{o}_3} \frac{\partial \mathbf{o}_3}{\partial \mathbf{h}_3} \boxed{\frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}} \frac{\partial \mathbf{h}_2}{\partial \mathbf{W}_r} + \frac{\partial \boldsymbol{\varepsilon}_3}{\partial \mathbf{o}_3} \frac{\partial \mathbf{o}_3}{\partial \mathbf{h}_3} \frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_r}$$

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

RNN gradients

$$\frac{\partial \boldsymbol{\varepsilon}_3}{\partial \mathbf{o}_3} = \frac{\partial \boldsymbol{\varepsilon}_3}{\partial \mathbf{o}_3} \frac{\partial \mathbf{o}_3}{\partial \mathbf{h}_3} \mathbf{W}_r \ \mathbf{W}_r \frac{\partial \mathbf{h}_1}{\partial \mathbf{W}_r} + \frac{\partial \boldsymbol{\varepsilon}_3}{\partial \mathbf{o}_3} \frac{\partial \mathbf{o}_3}{\partial \mathbf{h}_3} \mathbf{W}_r \frac{\partial \mathbf{h}_2}{\partial \mathbf{W}_r} + \frac{\partial \boldsymbol{\varepsilon}_3}{\partial \mathbf{o}_3} \frac{\partial \mathbf{o}_3}{\partial \mathbf{h}_3} \frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_r}$$

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

*** Ignoring the activation functions.**

RNN gradients

$$\mathbf{W}_r^n$$

RNN gradients

$$\mathbf{W}^{n} = (\mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^{-1})^{n}$$

RNN gradients

$$W^{n} = V\Lambda^{n}V^{-1}$$

RNN gradients

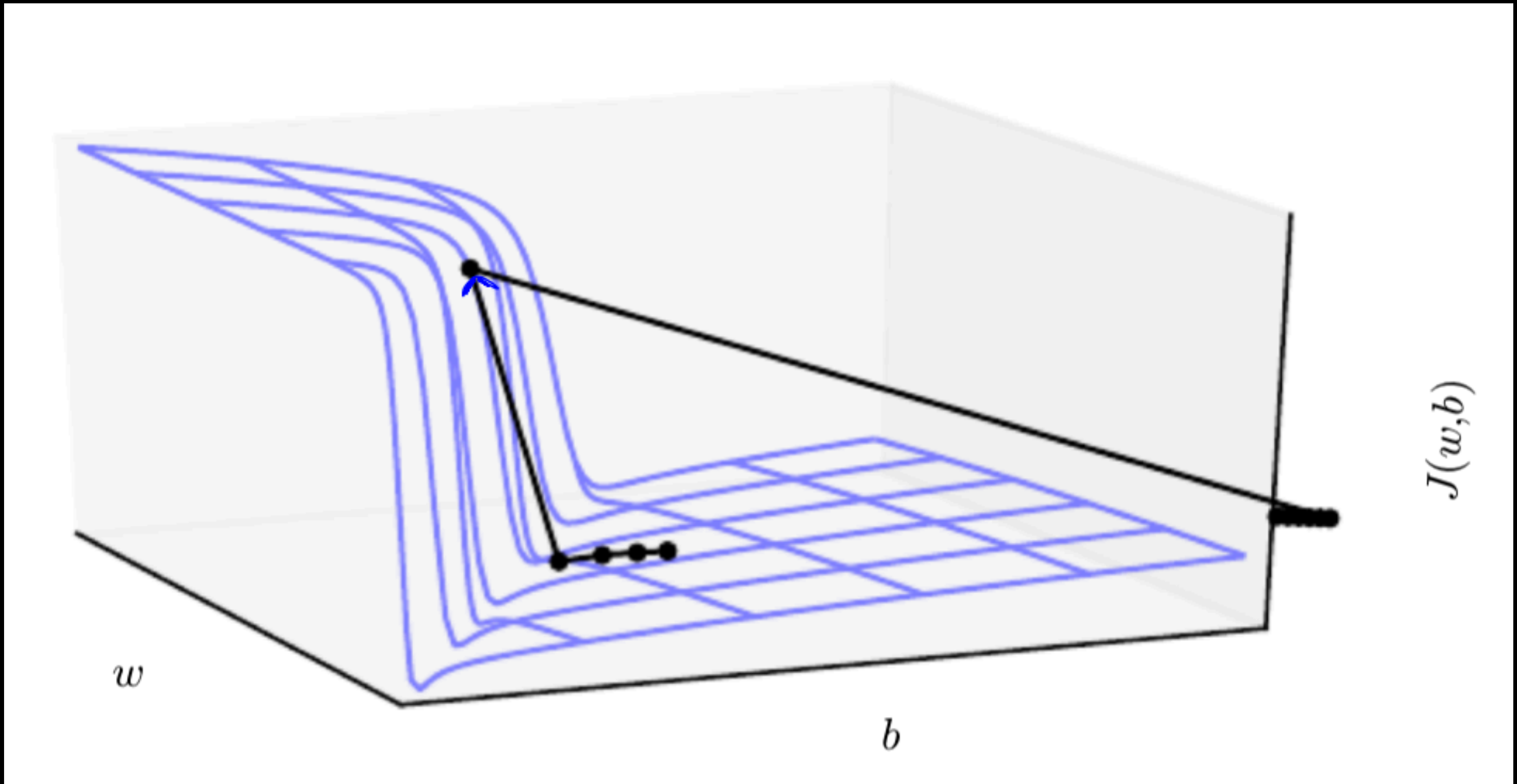
$$W^n = V \begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 \\ 0 & 0 & \lambda_3 & 0 \\ 0 & 0 & 0 & \lambda_4 \end{bmatrix}^n V^{-1}$$

RNN gradients

$$W^n = V \boxed{\begin{matrix} \lambda_1^n & 0 & 0 & 0 \\ 0 & \lambda_2^n & 0 & 0 \\ 0 & 0 & \lambda_3^n & 0 \\ 0 & 0 & 0 & \lambda_4^n \end{matrix}} V^{-1}$$

$\lambda_n > 1$ causes that part of the gradient to explode.
$\lambda_n < 1$ causes that part of the gradient to to vanish.

RNN gradients



LSTMs

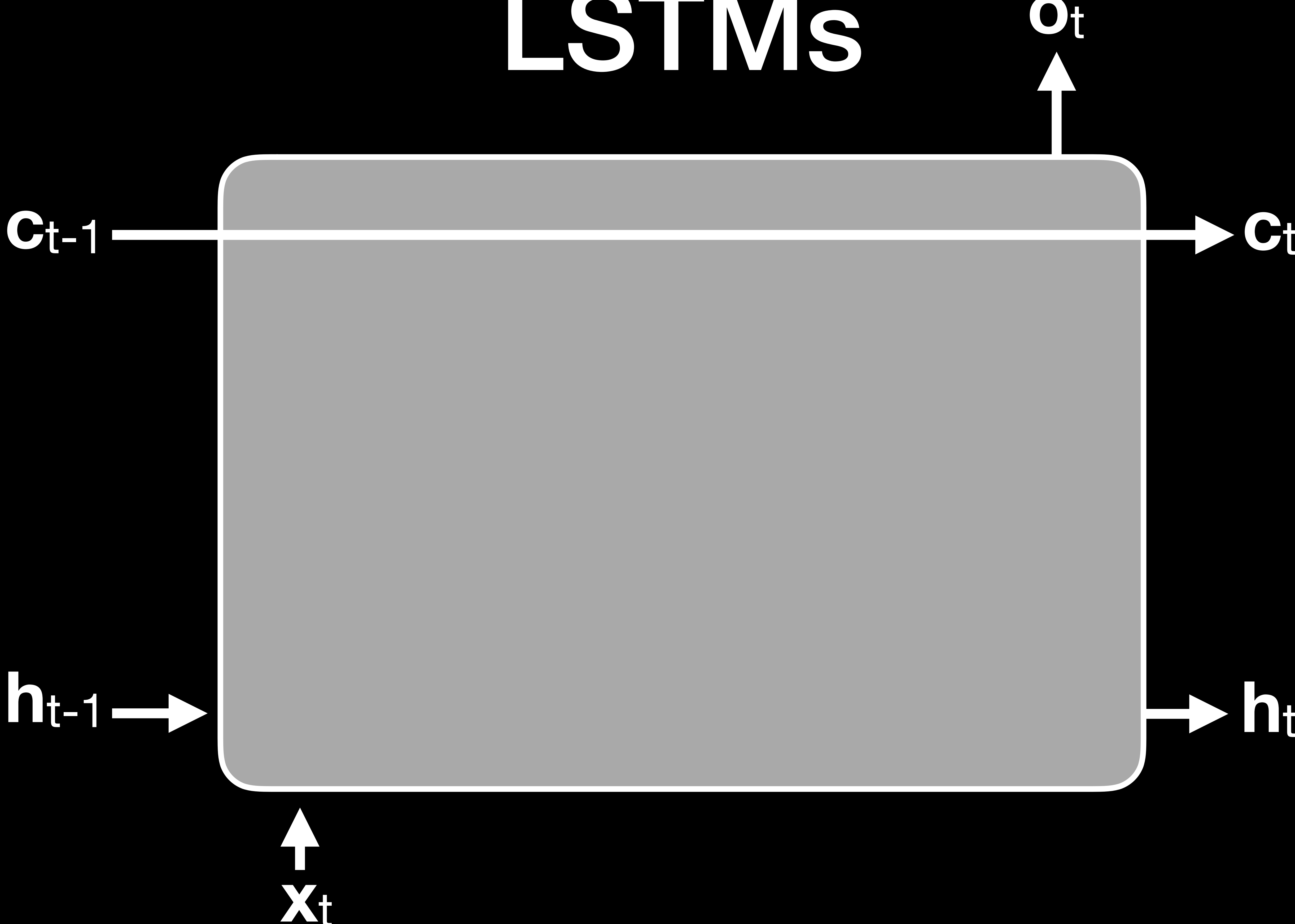
LSTMs

$$\begin{bmatrix} 7 \\ 5 \\ 10 \\ 9 \end{bmatrix} \times \begin{bmatrix} 0.5 \\ 0 \\ 0.2 \\ 1 \end{bmatrix} = \begin{bmatrix} 3.5 \\ 0 \\ 2 \\ 9 \end{bmatrix}$$

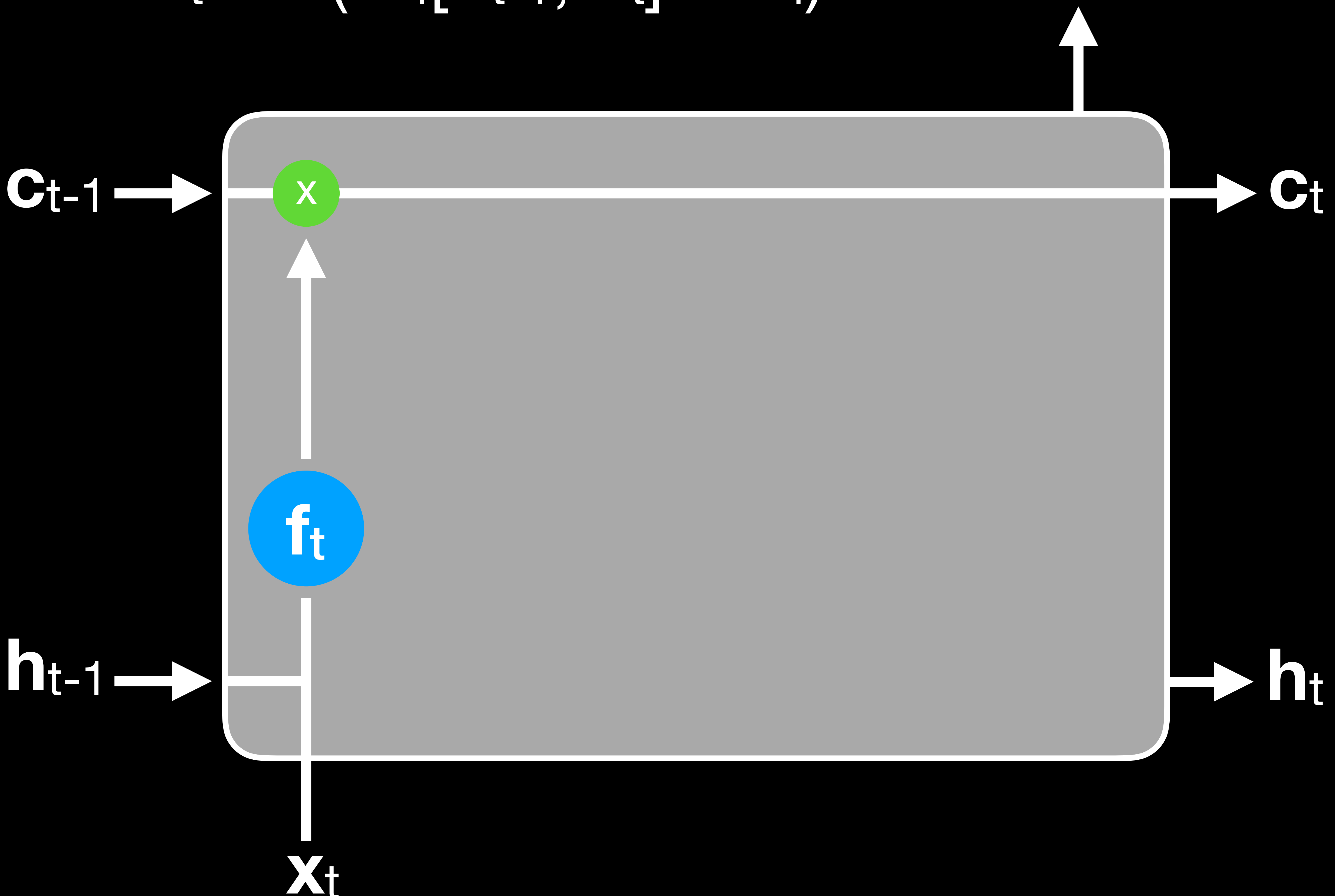
LSTMs

$$\boxed{\begin{matrix} 0.5 \\ 0 \\ 0.2 \\ 1 \end{matrix}} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

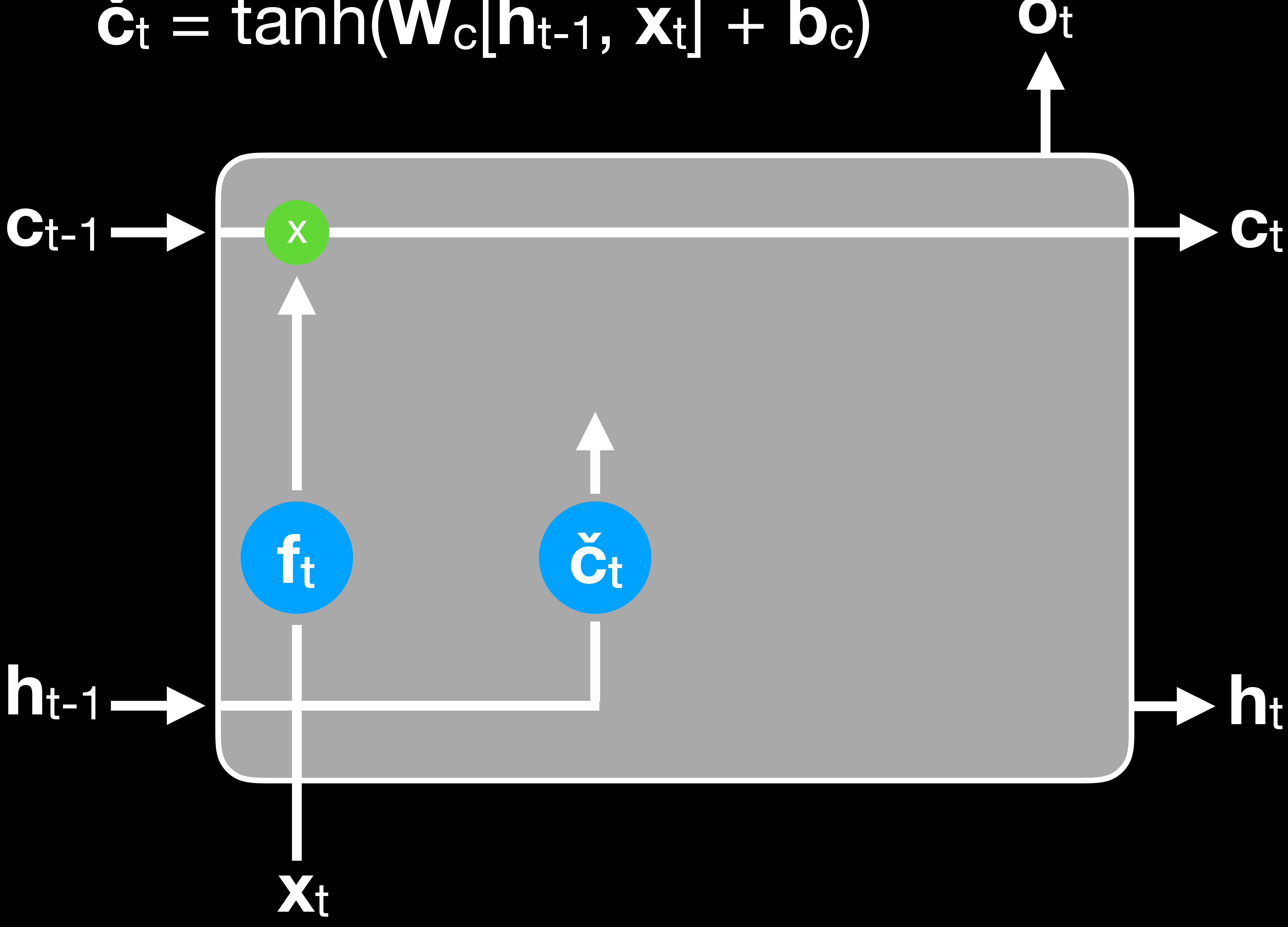
LSTMs



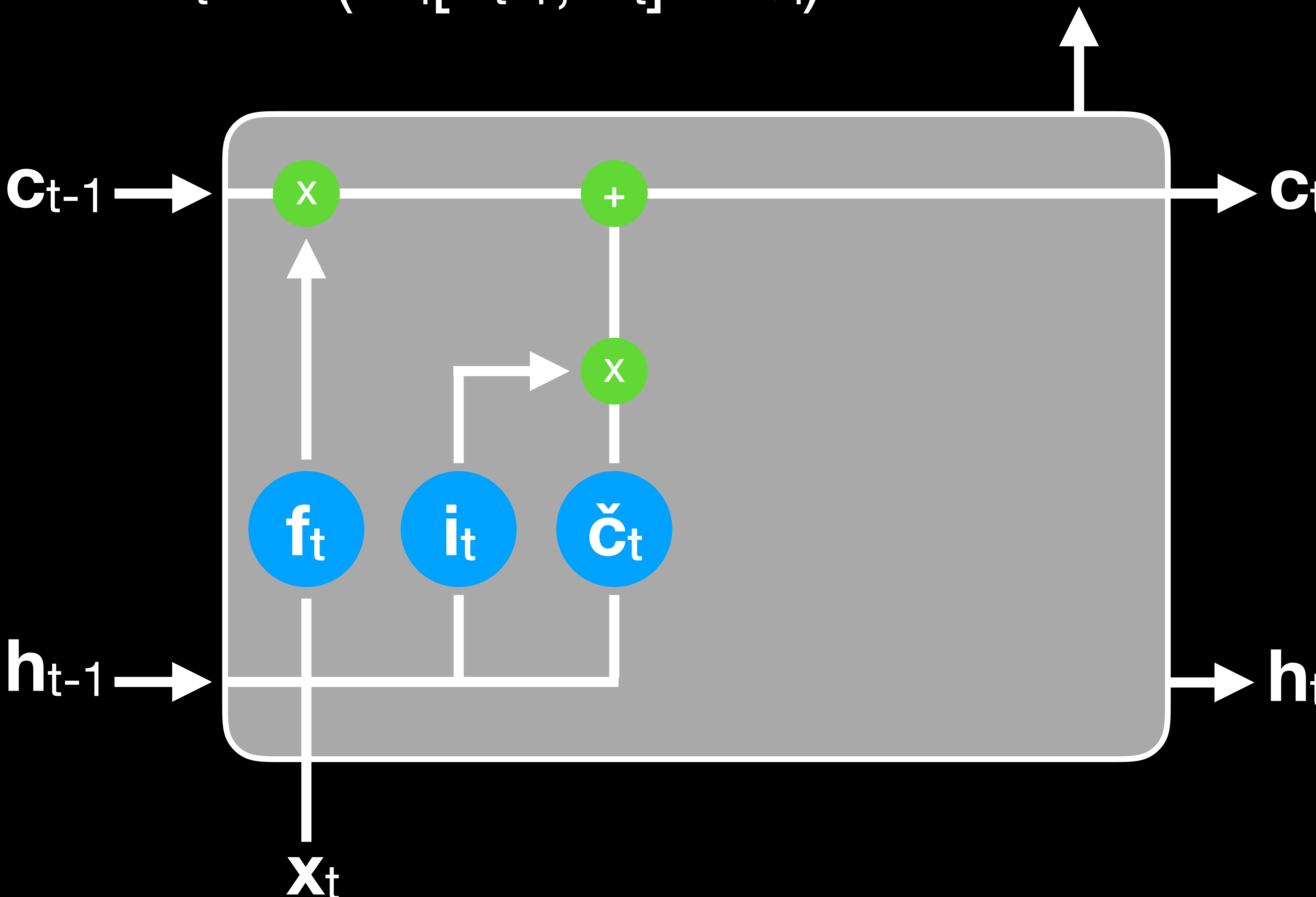
$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$



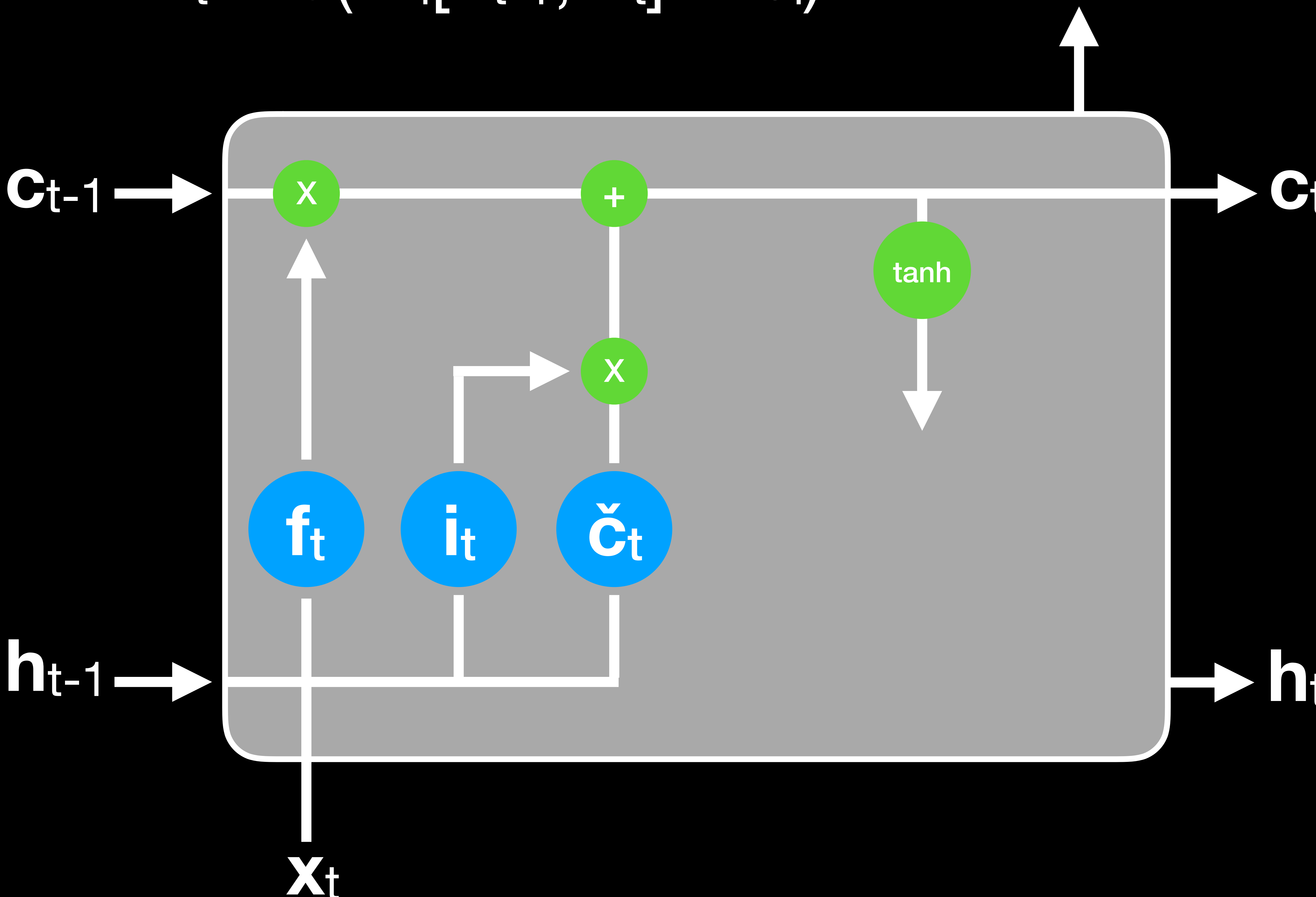
$$\check{c}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c)$$



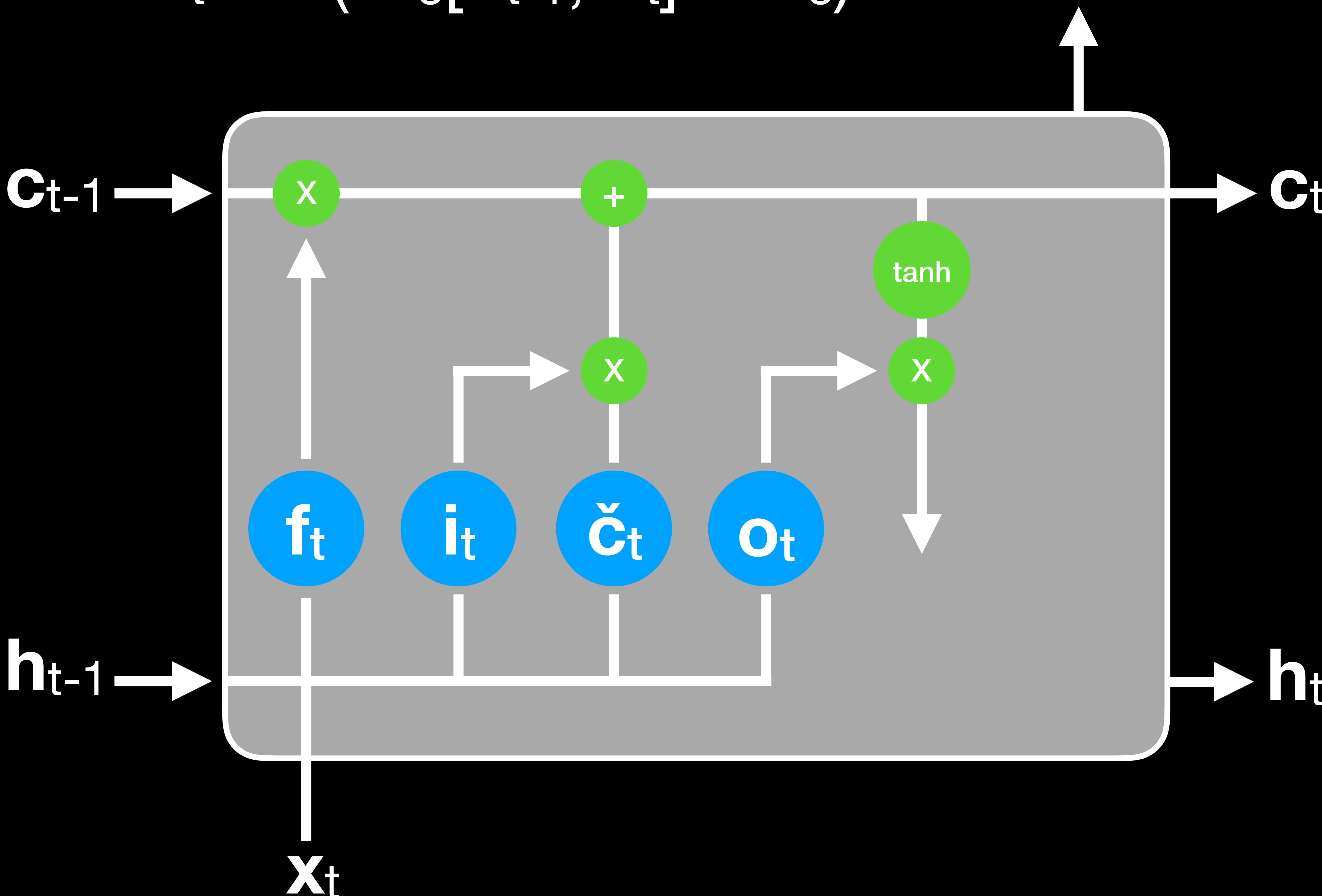
$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$



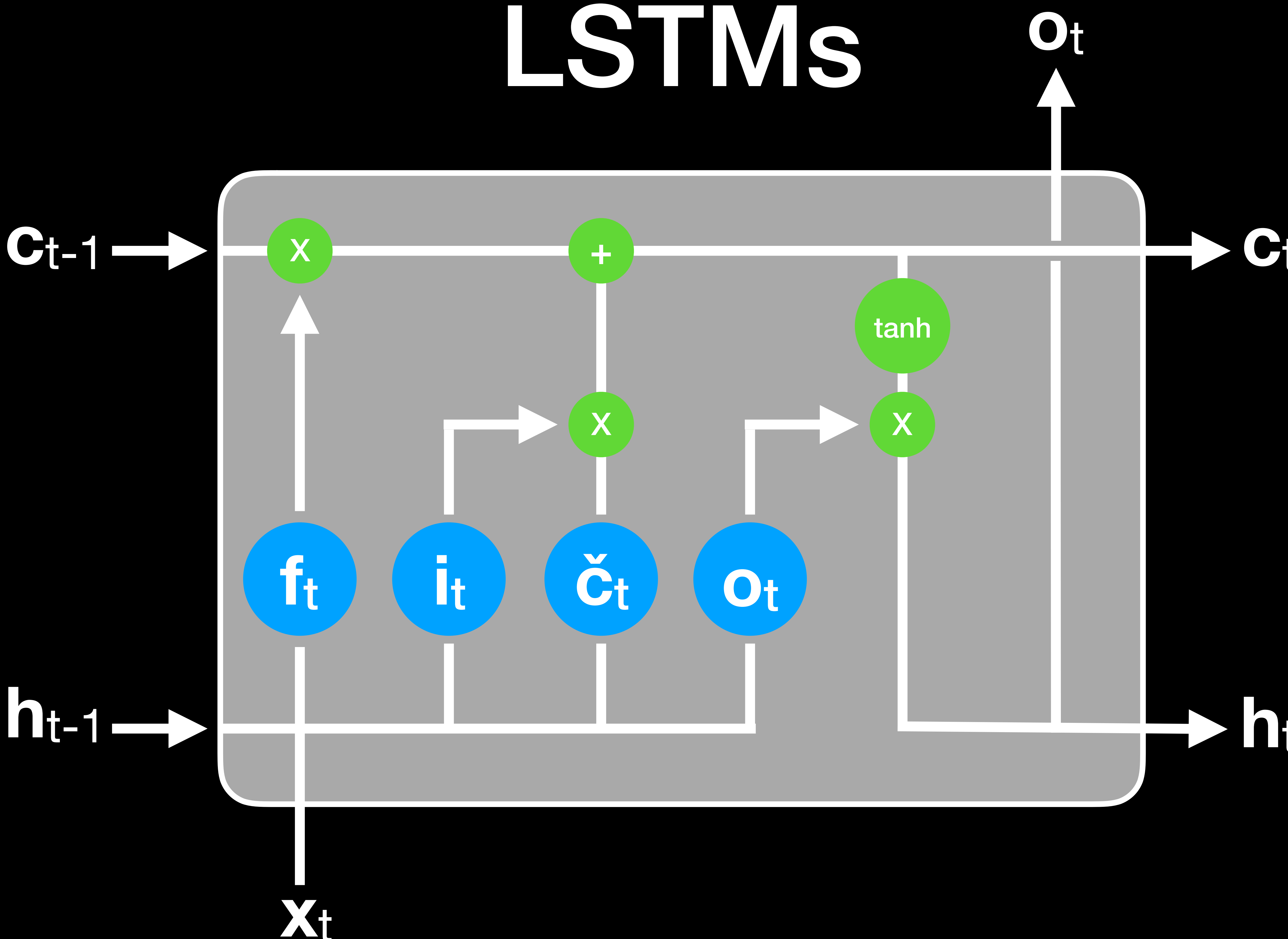
$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$



$$\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o)$$



LSTMs



How do LSTMs solve the problems of vanilla RNNs?

Going forwards

**The cell state is never squashed or scaled—
information is only lost via the forget gate.**

$$\mathbf{c}_t = \mathbf{f}_t \times \mathbf{c}_{t-1} + \mathbf{i}_t \times \check{\mathbf{c}}_t$$

Going backwards

In the vanilla RNN, it was the repeated multiplication of the hidden state by W_h that led to powers of W_h appearing in W_h's derivative with respect to error.

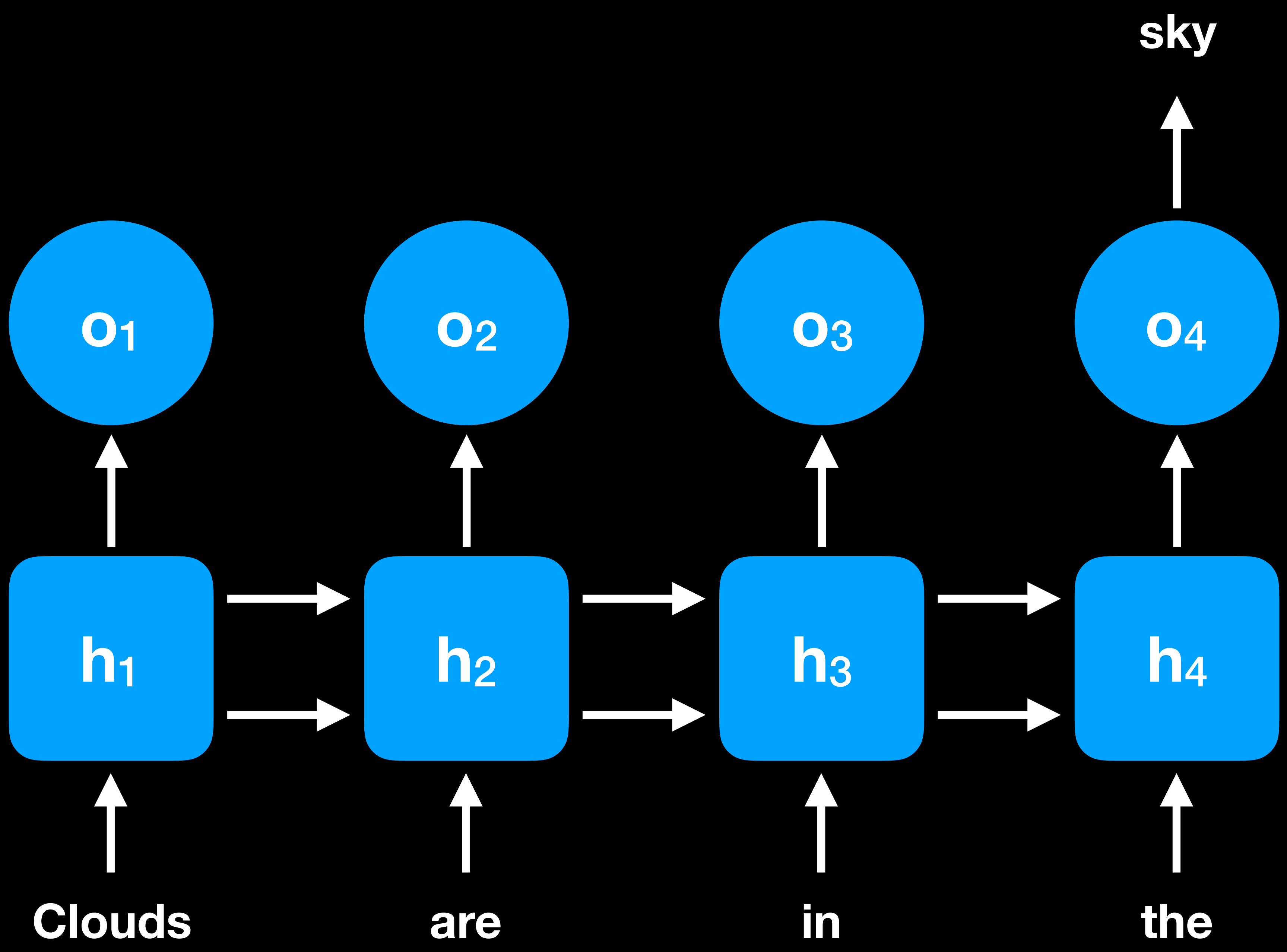
In the LSTM, you're still reusing the same matrices over and over, but they're not directly applied to their output at a previous step: everything is intermediated by the cell state. Intuitively, this makes it harder for powers of the W_c to appear in the gradients, though not impossible[1]. In those cases, gradient clipping can be used to prevent exploding gradients. (Vanishing gradients are less of an issue. Important past info can be regularly written out of the cell state, and so can impact W_c's gradient even if some of the earlier terms of the gradient relating to that info vanish.)

$$\mathbf{c}_t = \mathbf{f}_t \times \mathbf{c}_{t-1} + \mathbf{i}_t \times \check{\mathbf{c}}_t$$

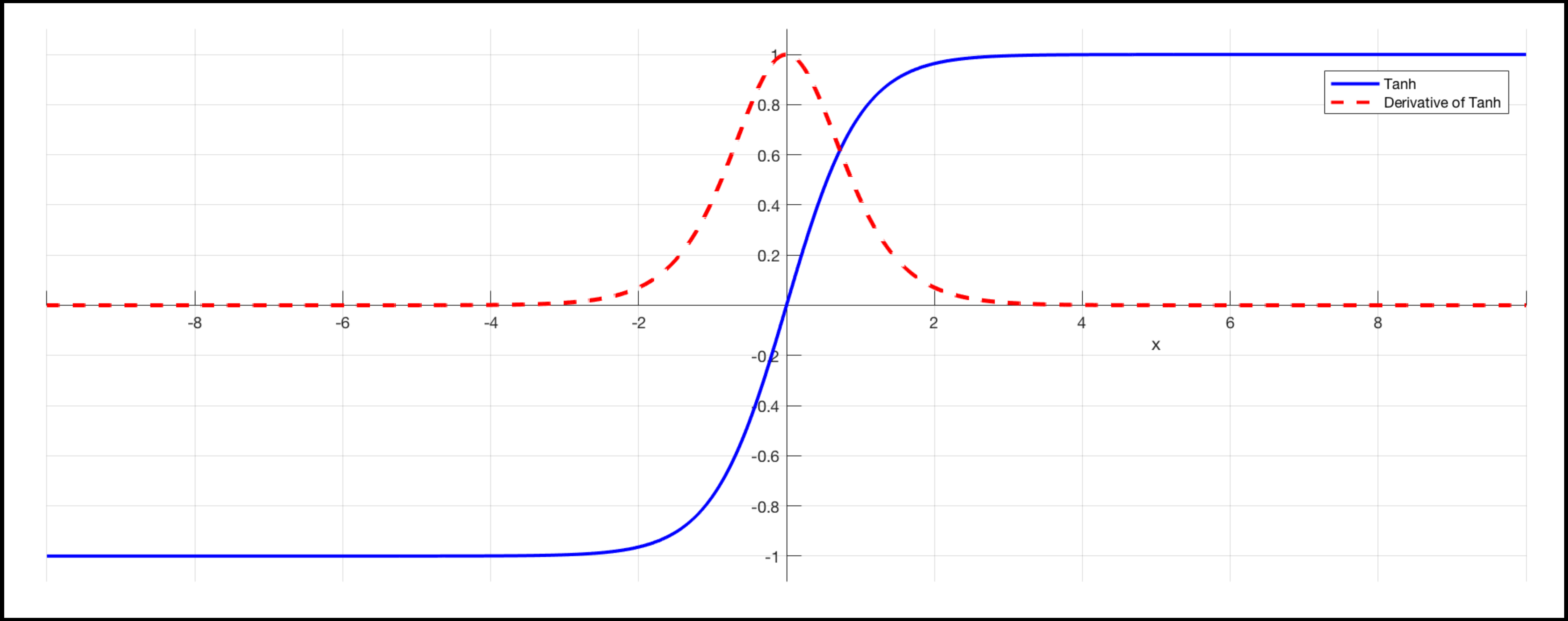
$$\mathbf{h}_t = \mathbf{o}_t \times \tanh(\mathbf{c}_t)$$

1. Imagine a pathological case where the forget gate was somehow set to always forget and the output gate was somehow set to always output everything—this would basically give you a vanilla RNN.

LSTMs



tanh(x)



sigmoid σ(x)