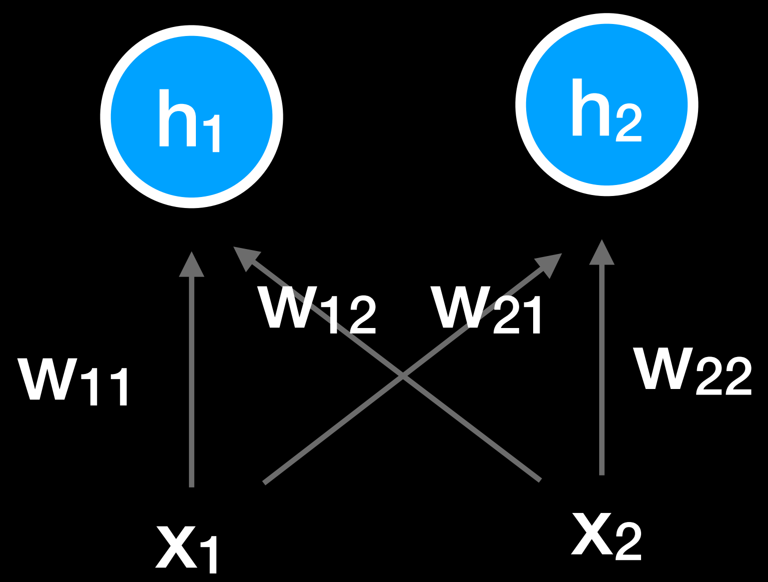
Recurrent neural networks

Nicholas Beshouri — nbeshouri@gmail.com

Neural nets review



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

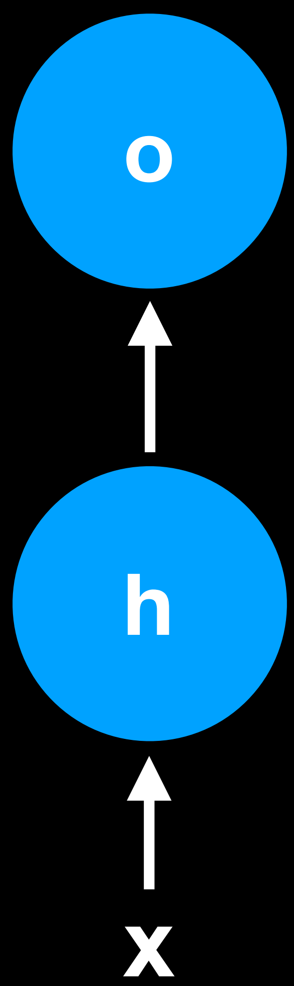
$$\boxed{\begin{matrix} \mathrm{foo}(w_{11}x_1 + w_{12}x_2 + b_1) \\ \mathrm{foo}(w_{21}x_1 + w_{22}x_2 + b_2) \end{matrix}} = \mathrm{foo}\left[\boxed{\begin{matrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{matrix}} \boxed{\begin{matrix} x_1 \\ x_2 \end{matrix}} + \boxed{\begin{matrix} b_1 \\ b_2 \end{matrix}}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

Why do we need RNNs?

Motivating example

Clouds are in the _____

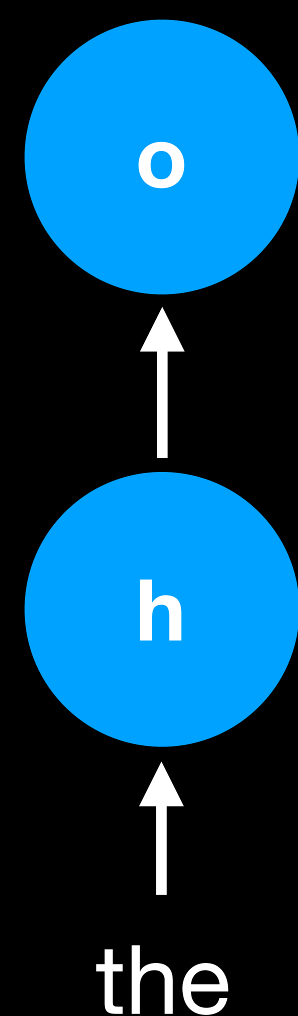
Motivating example



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

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

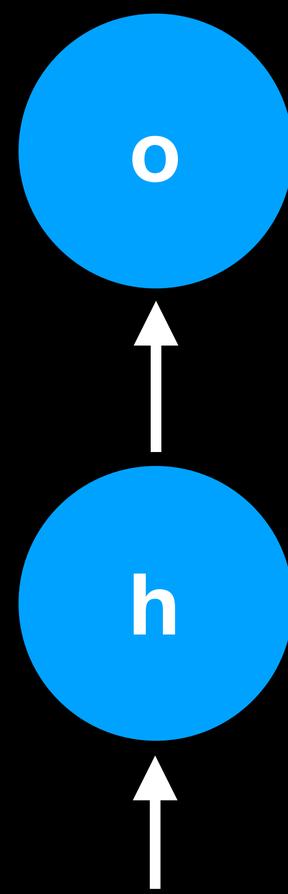
Motivating example



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

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

Motivating example

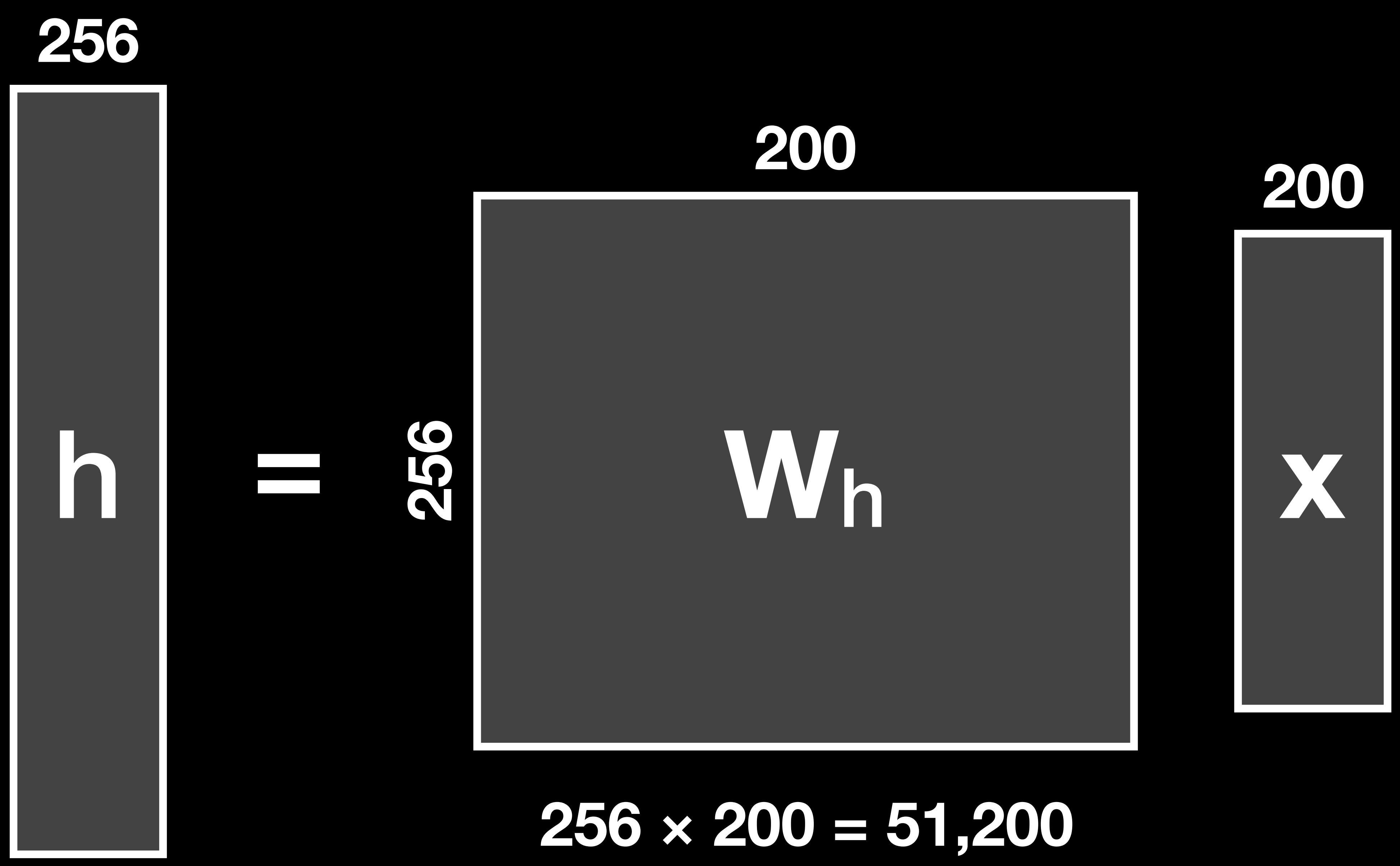


$= \mathrm{softmax}(\mathbf{W}_o\mathbf{h} + \mathbf{b}_o)$

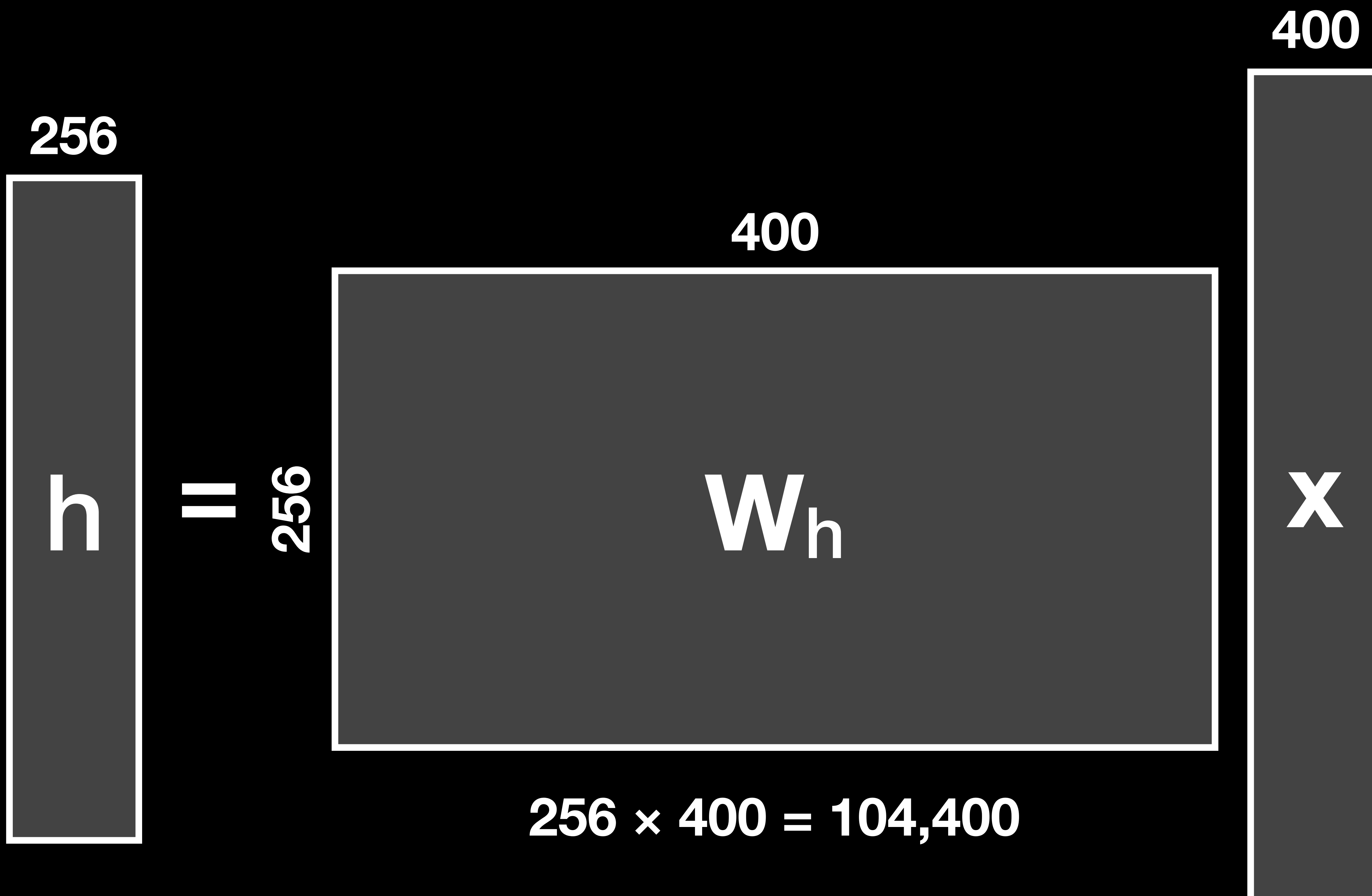
$= \mathrm{relu}(\mathbf{W}_h\mathbf{x} + \mathbf{b}_h)$

[in, the]

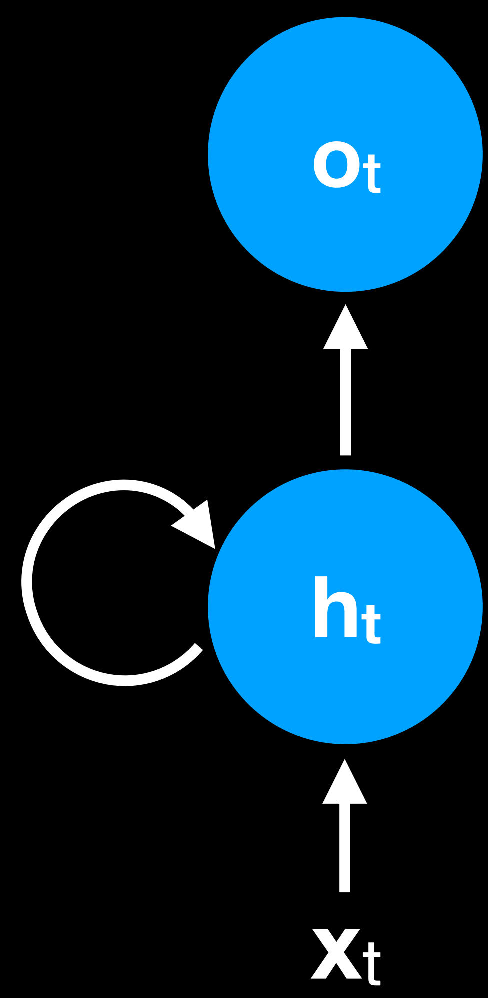
Motivating example



Motivating example



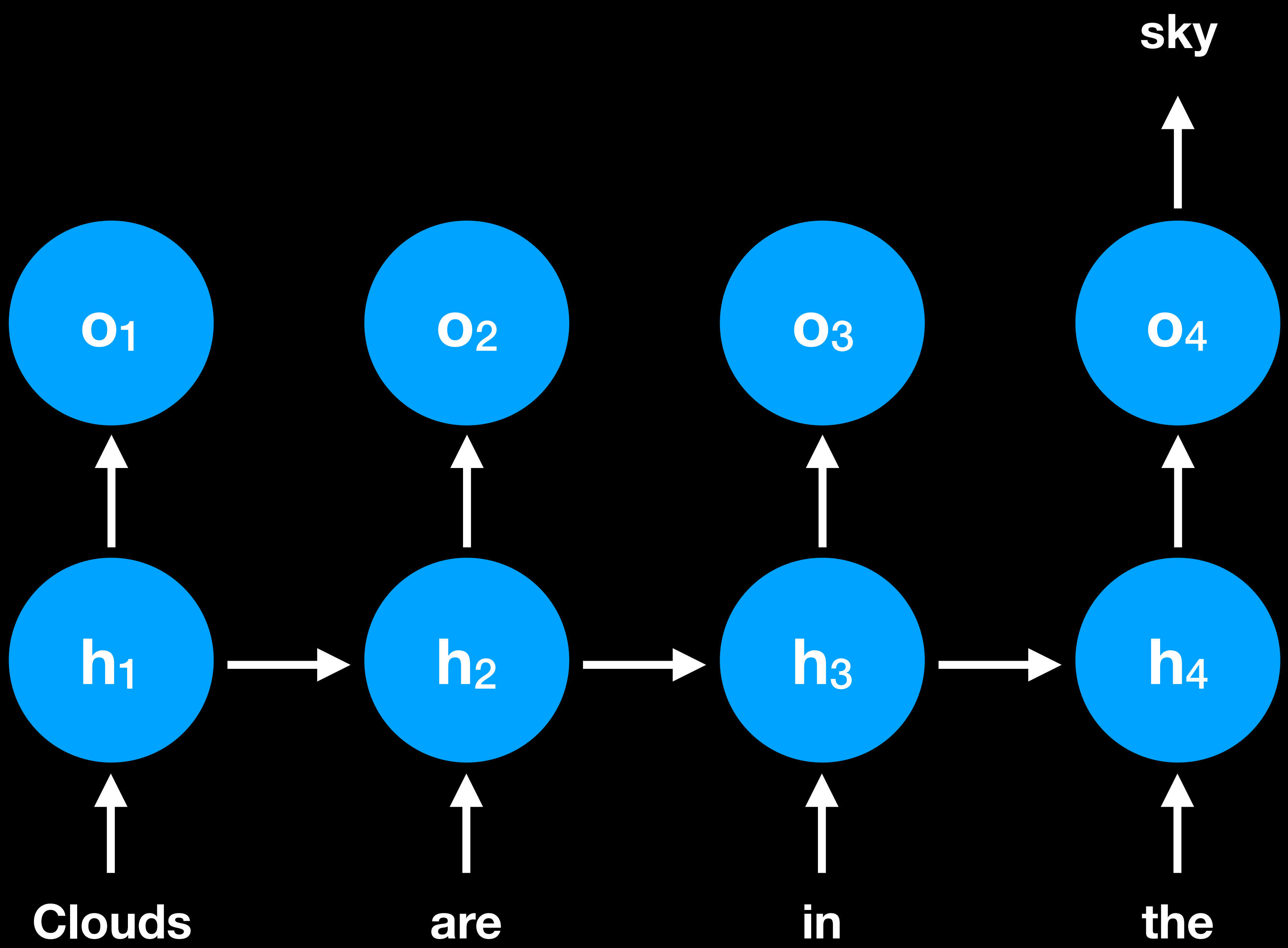
Vanilla RNNs



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

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

Vanilla RNNs



But they don't work...

The problems with vanilla RNNs

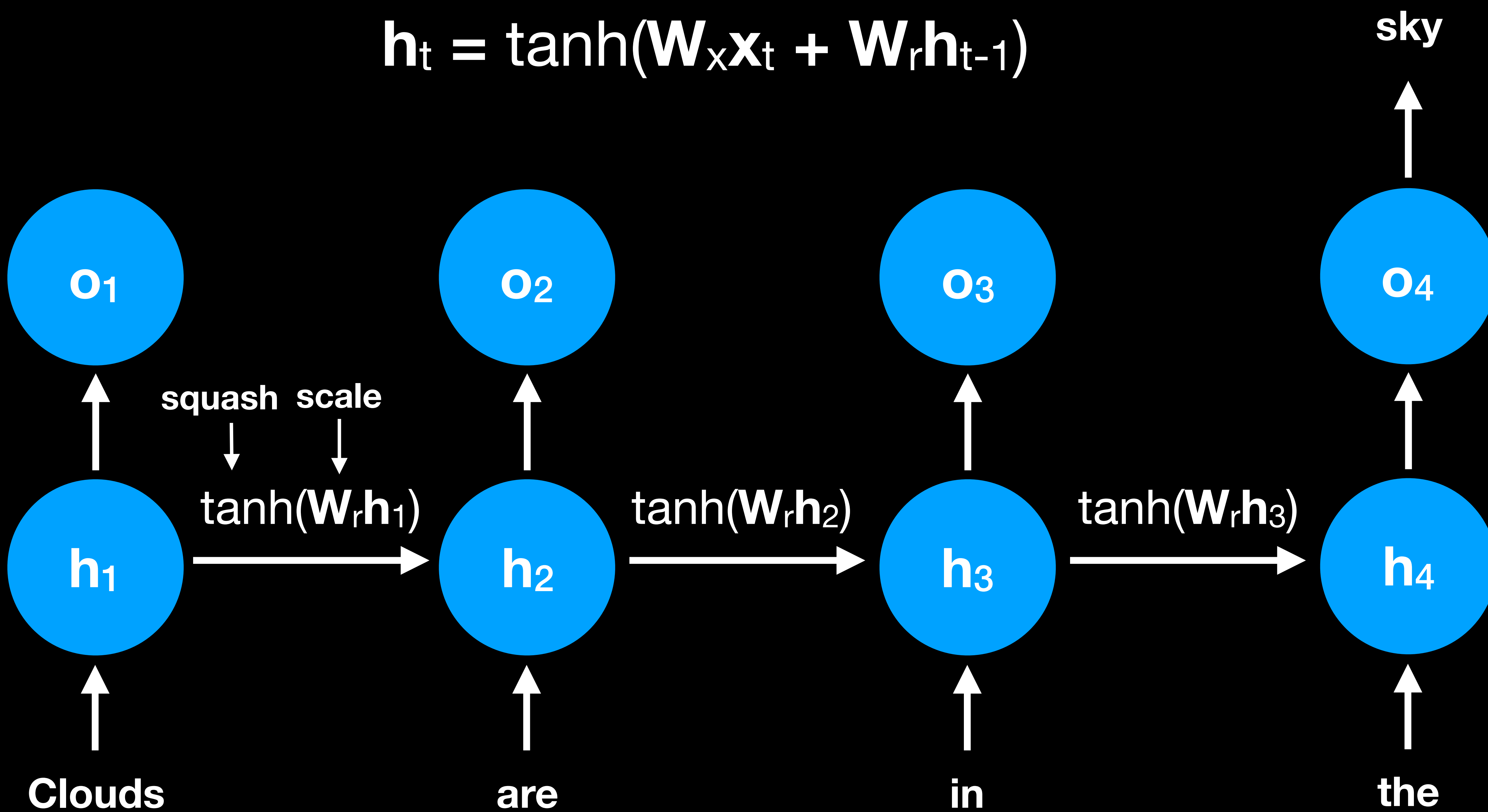
1. Problems going forwards: Signals from earlier in time become fainter and fainter.

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]:
```python
Wr = np.random.normal(0, 3, (2, 2))
Wr
```

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

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

scale

*squash

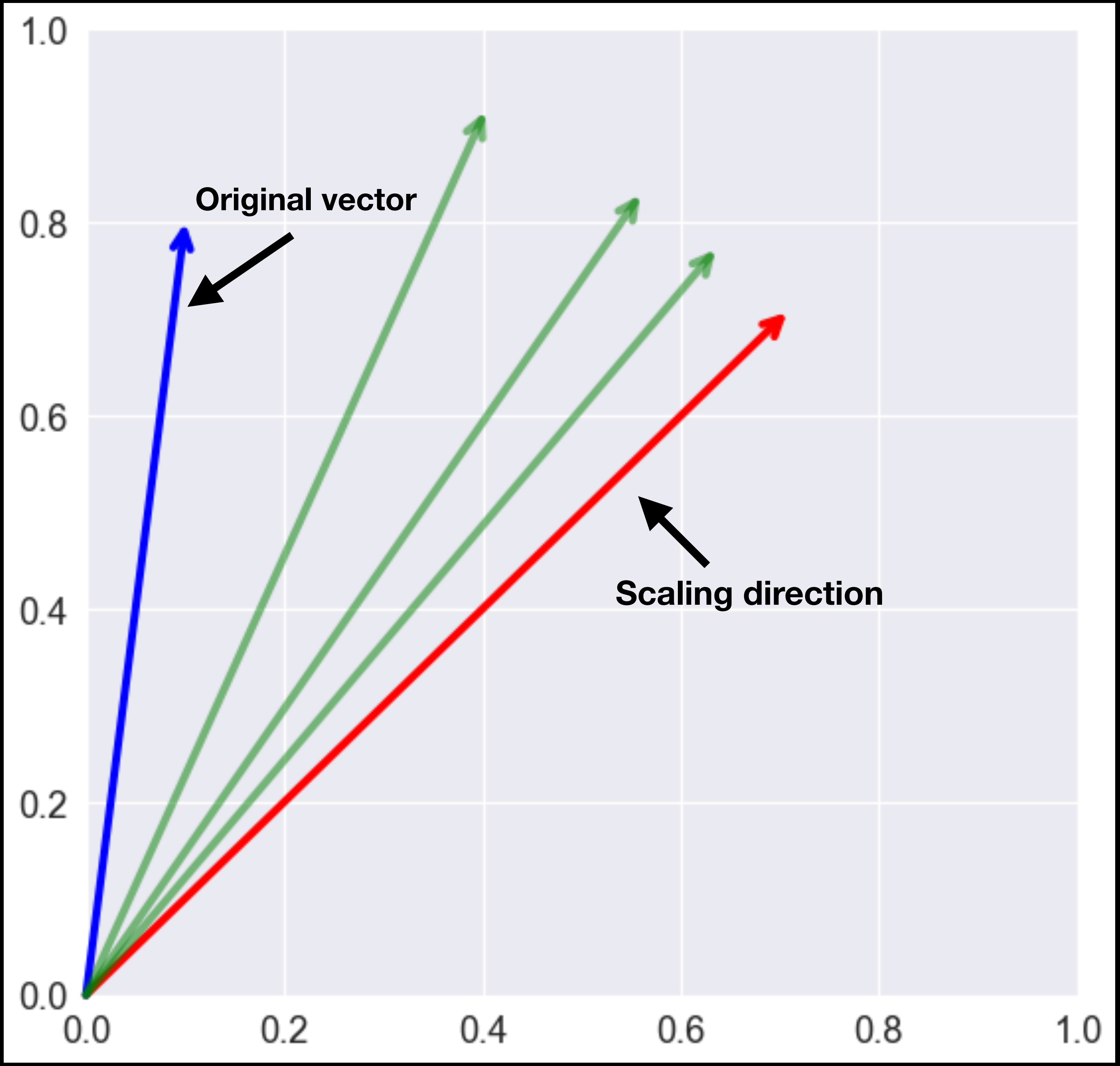
Out[25]:
```
array([ 0.12065226, -0.99269483])
```

Going forwards

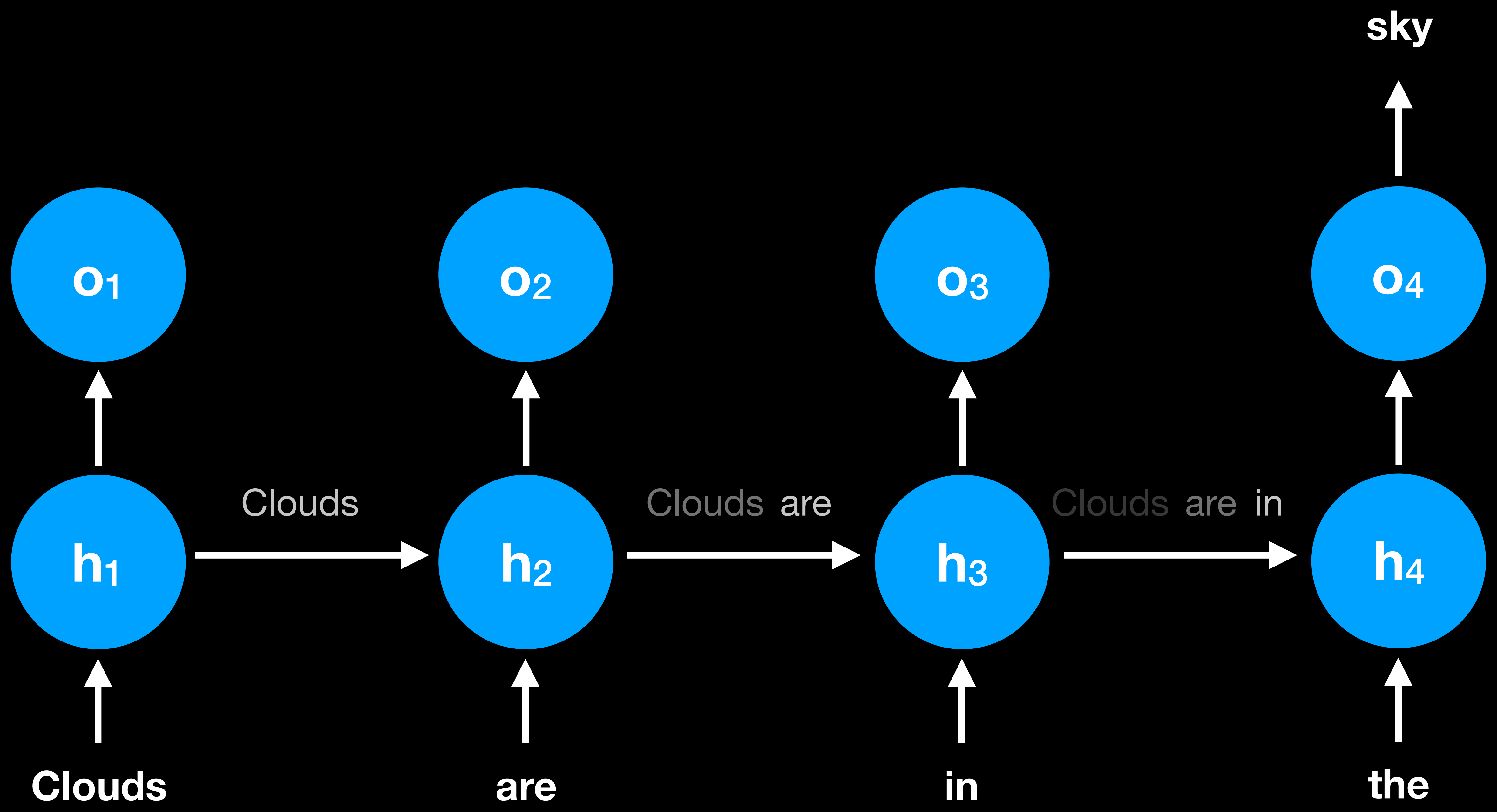
```
In [22]: Wr = np.random.normal(0, 3, (2, 2))
         Wr

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 forwards



Going backwards

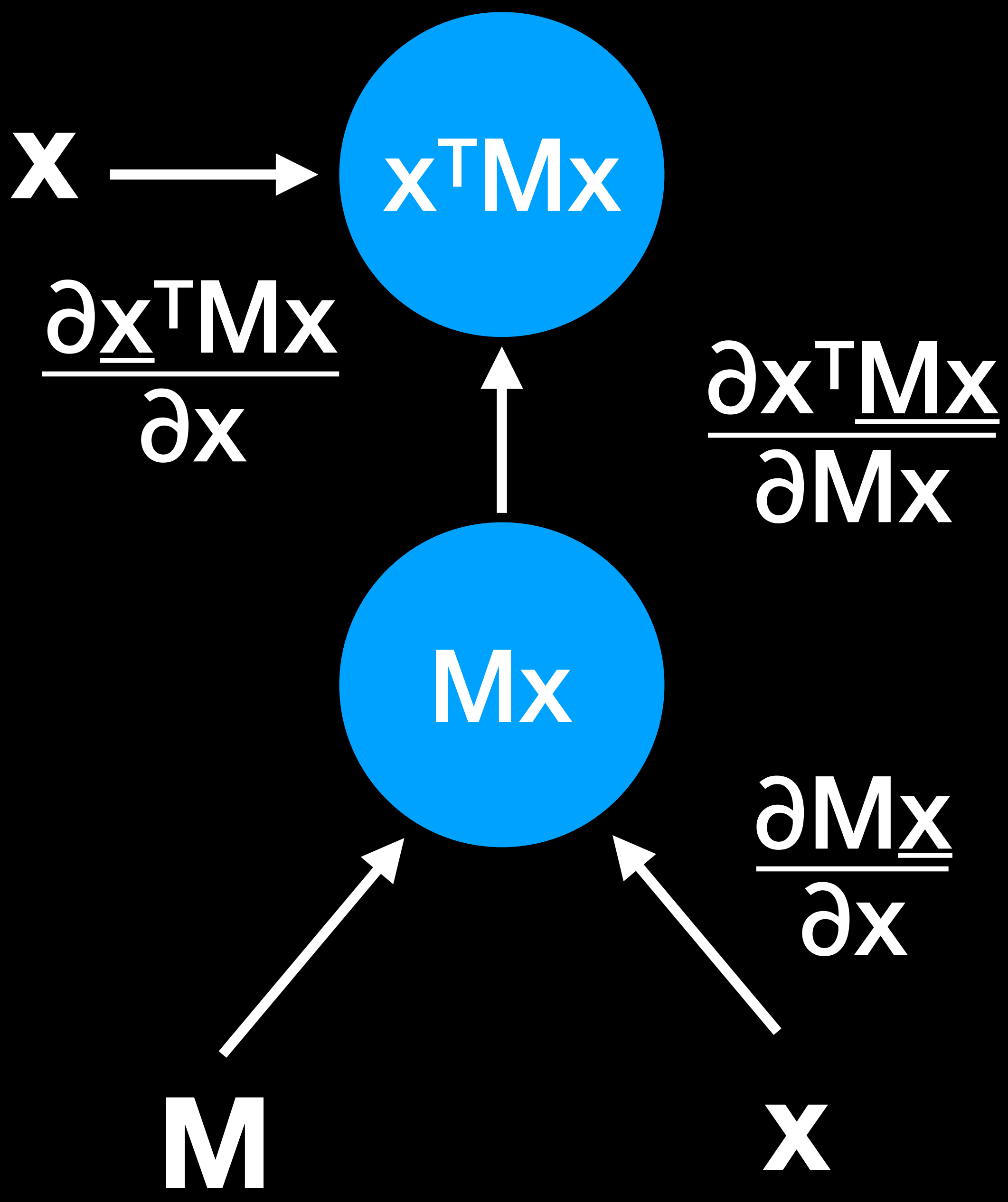
Backprop review

$$\mathbf{x}^\top \mathbf{M} \mathbf{x}$$

Backprop review

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

Backprop review



Backprop review

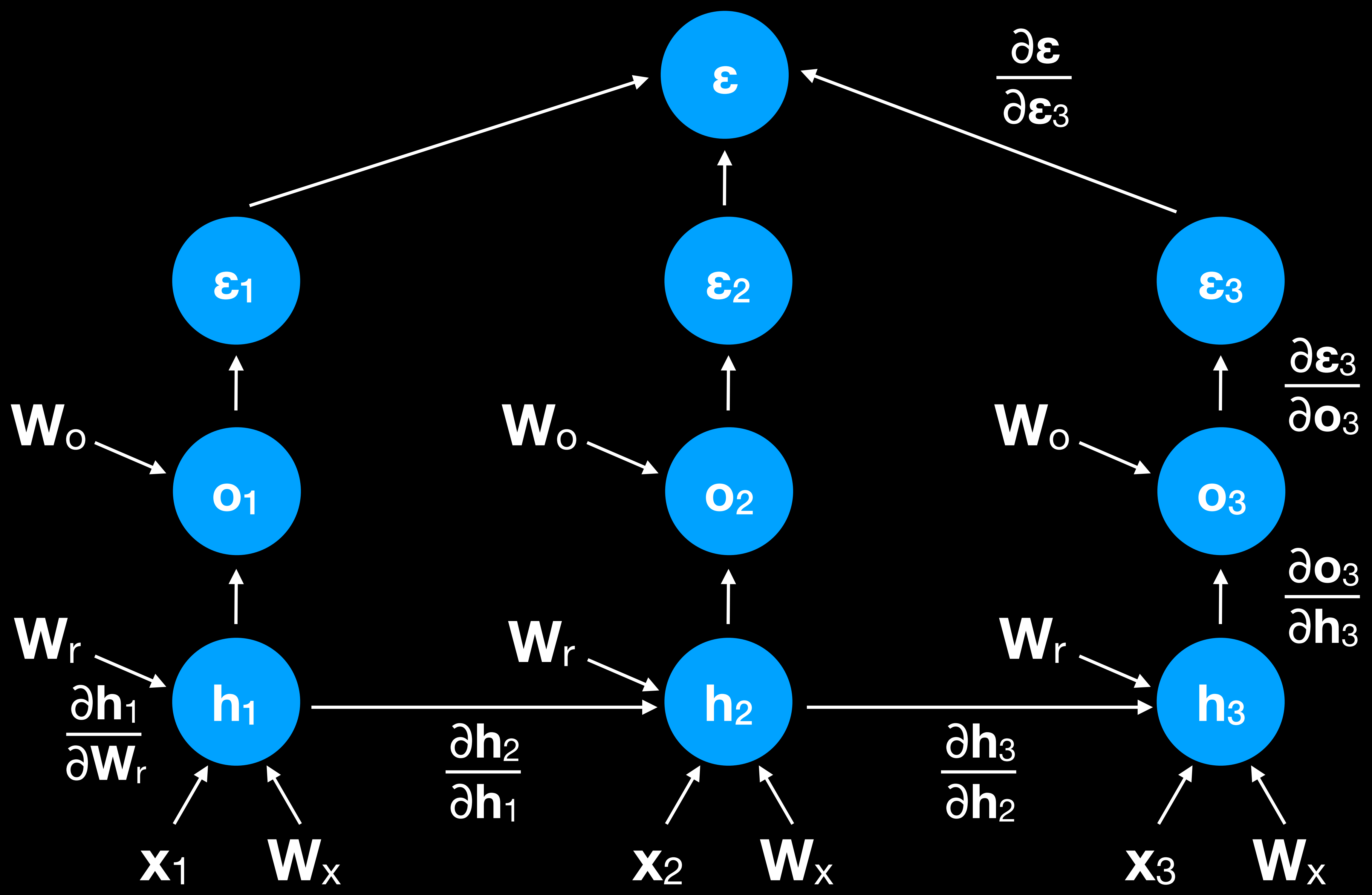
$$(\underline{x}^T M x)^T = x^T M^T \underline{x}$$

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

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

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

RNN gradients



RNN gradients

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

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

RNN gradients

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

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

*** Ignoring the activation functions.**

RNN gradients

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

RNN gradients

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

RNN gradients

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

RNN gradients

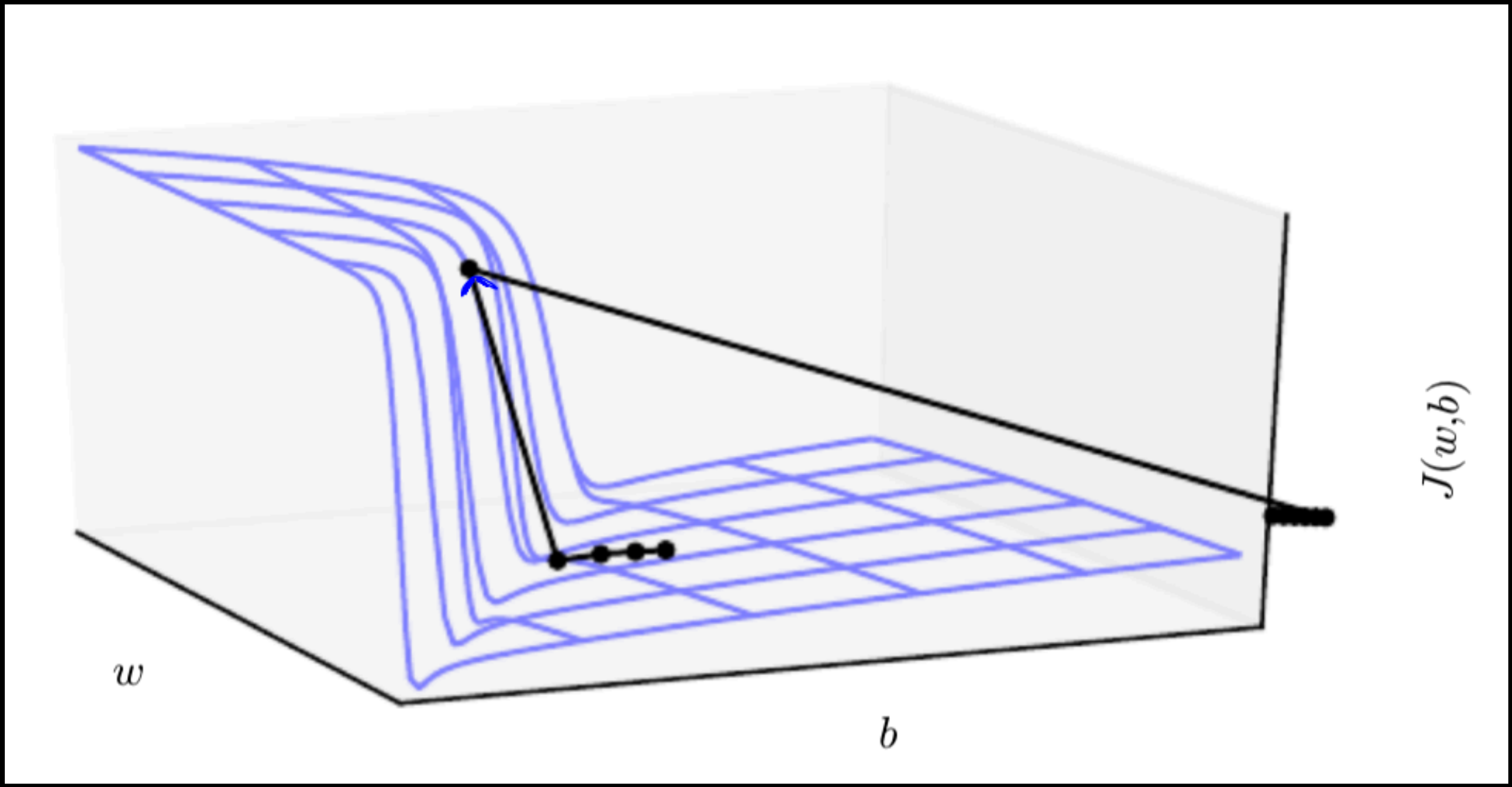
$$W^n = V \boxed{\begin{matrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 \\ 0 & 0 & \lambda_3 & 0 \\ 0 & 0 & 0 & \lambda_4 \end{matrix}}^{\,n} V^{-1}$$

RNN gradients

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

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

RNN gradients



LSTMs

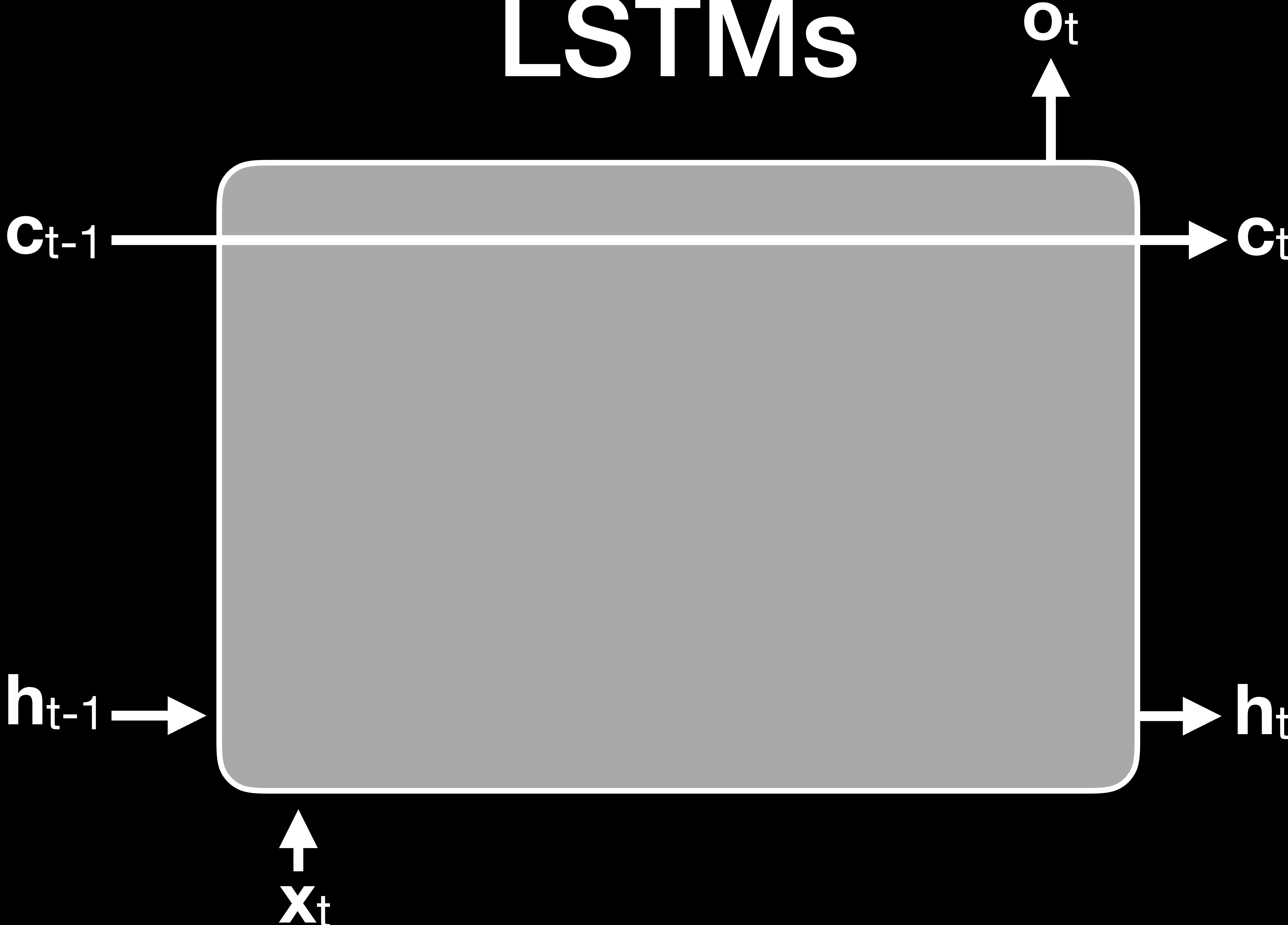
LSTMs

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

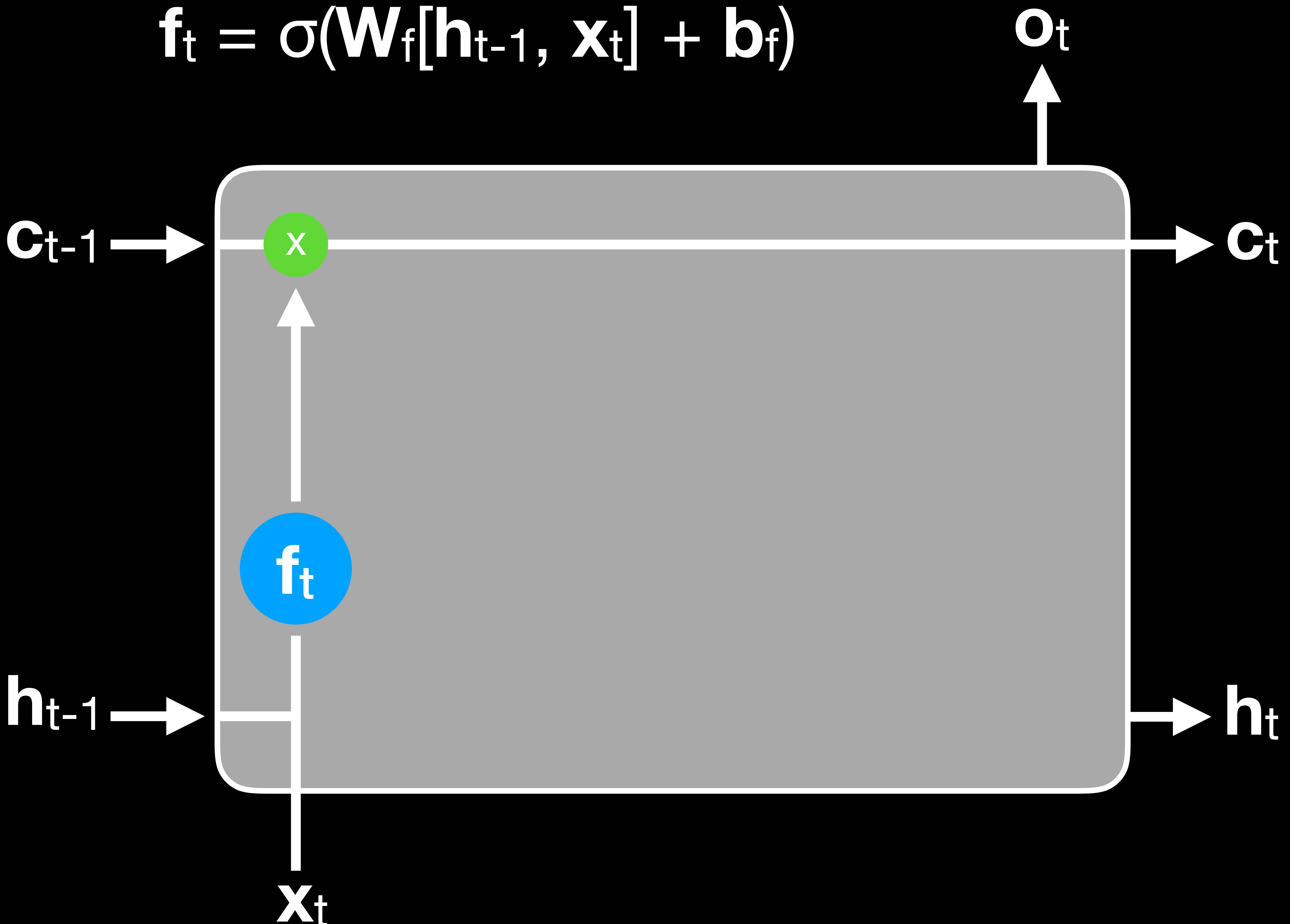
LSTMs

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

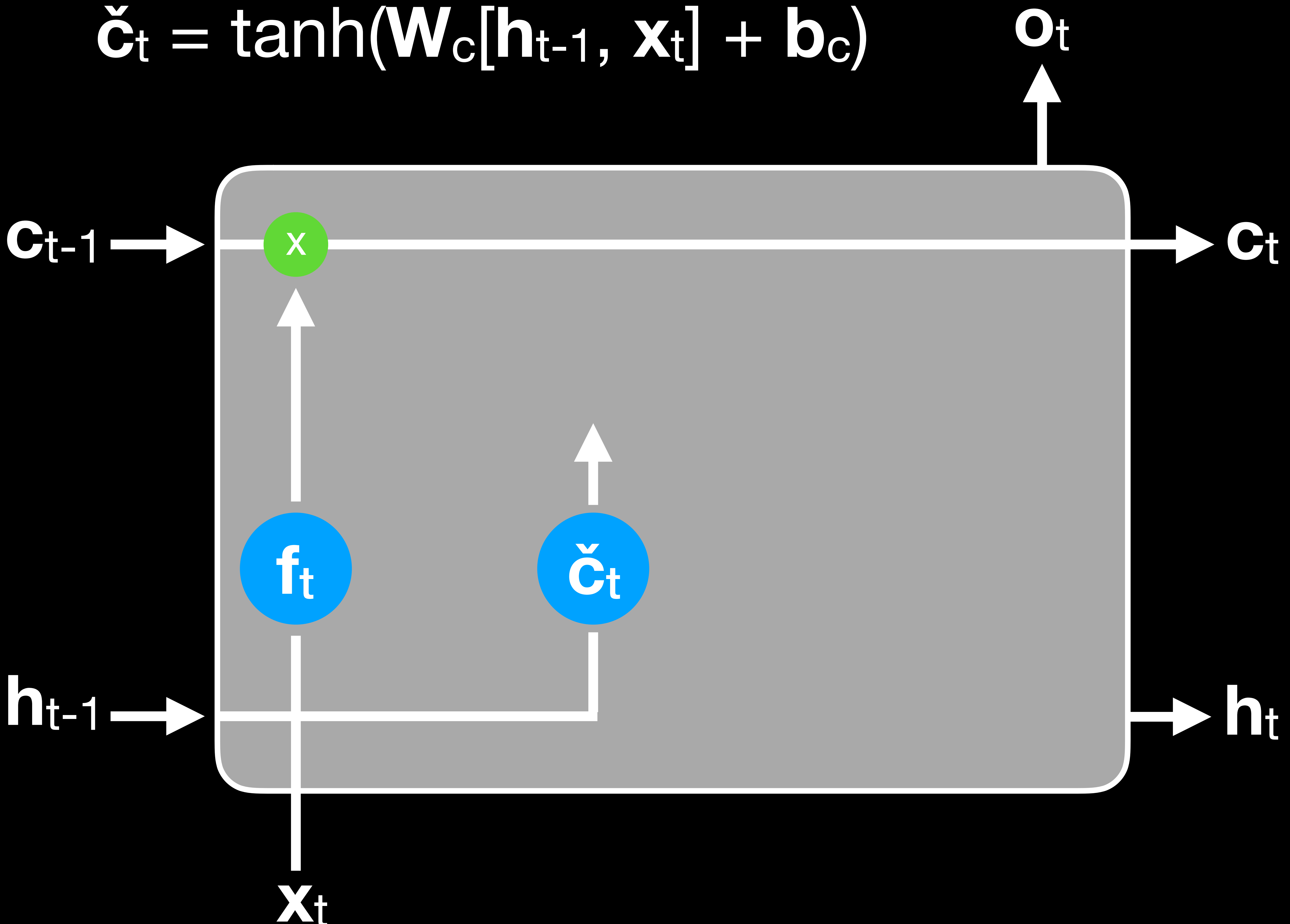
LSTMs



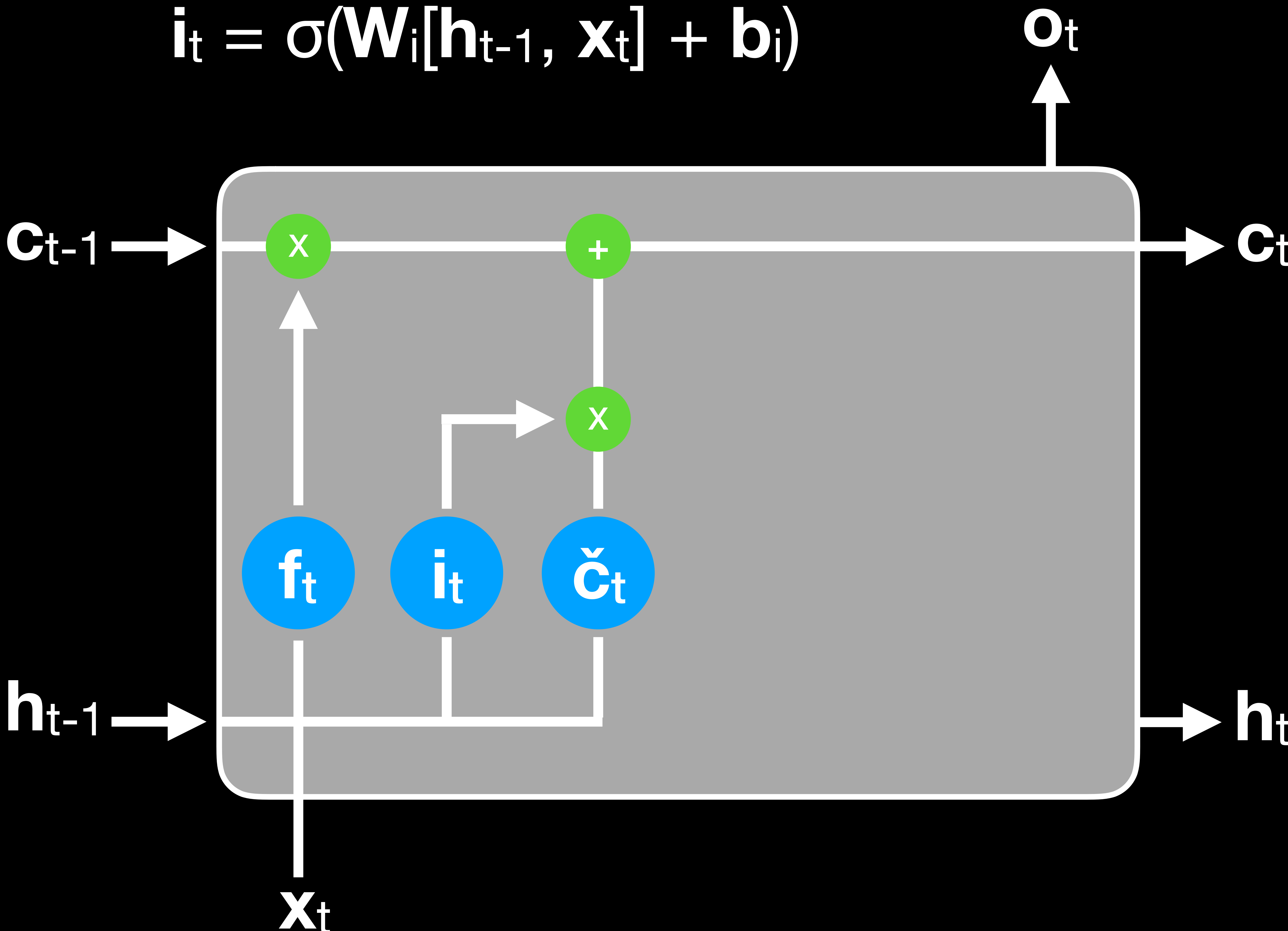
$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$



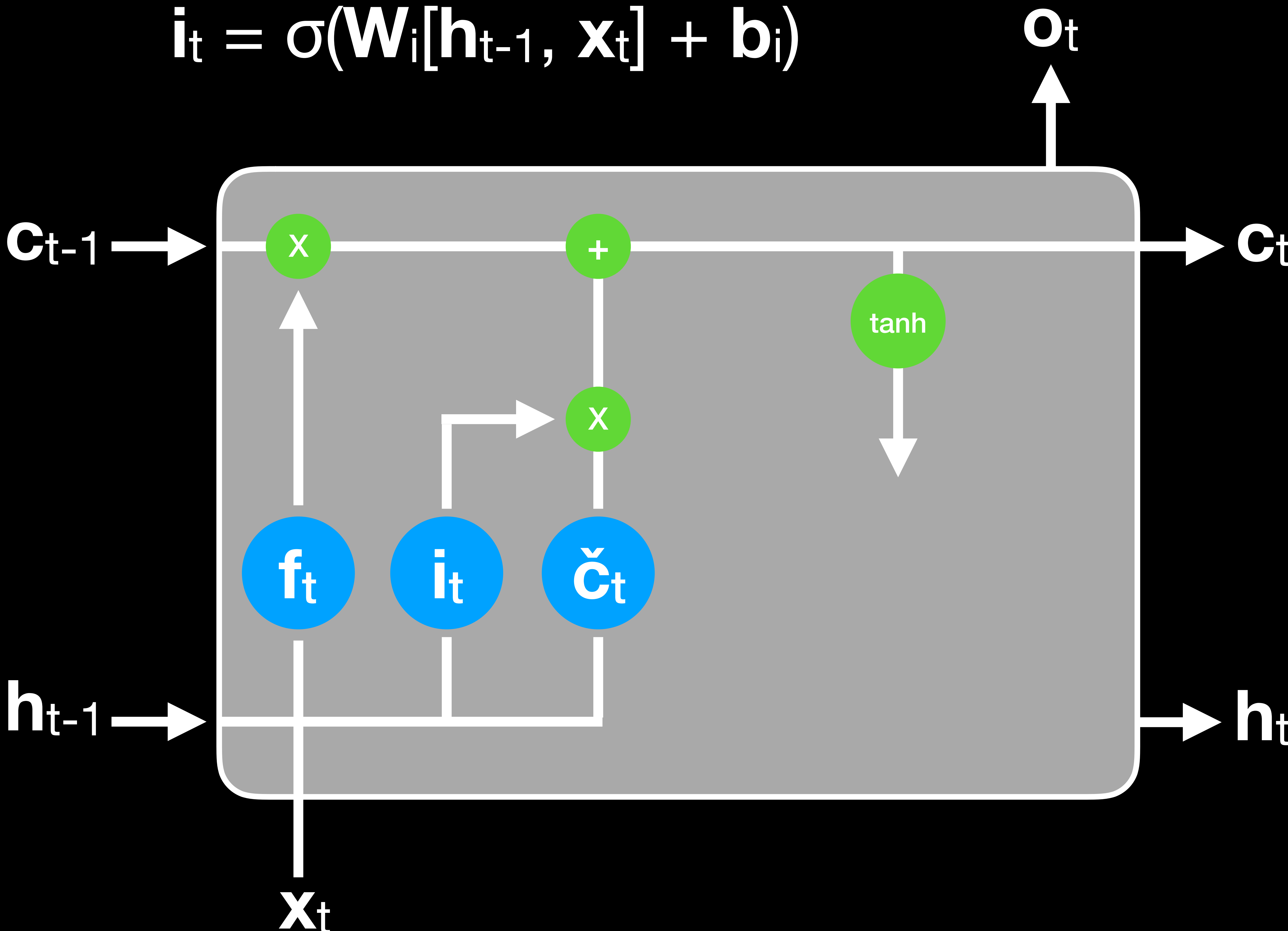
$$\check{c}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c)$$



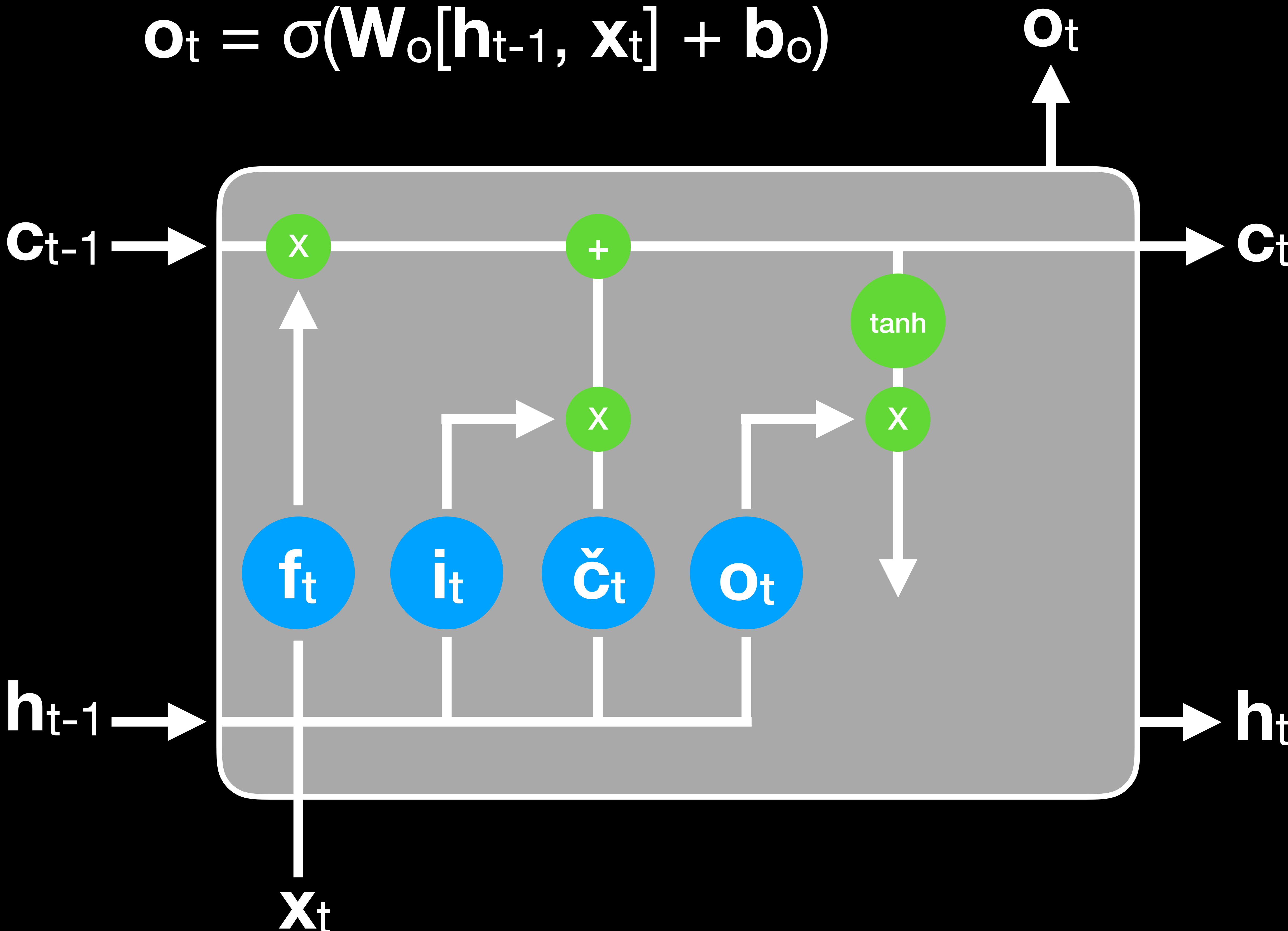
$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$



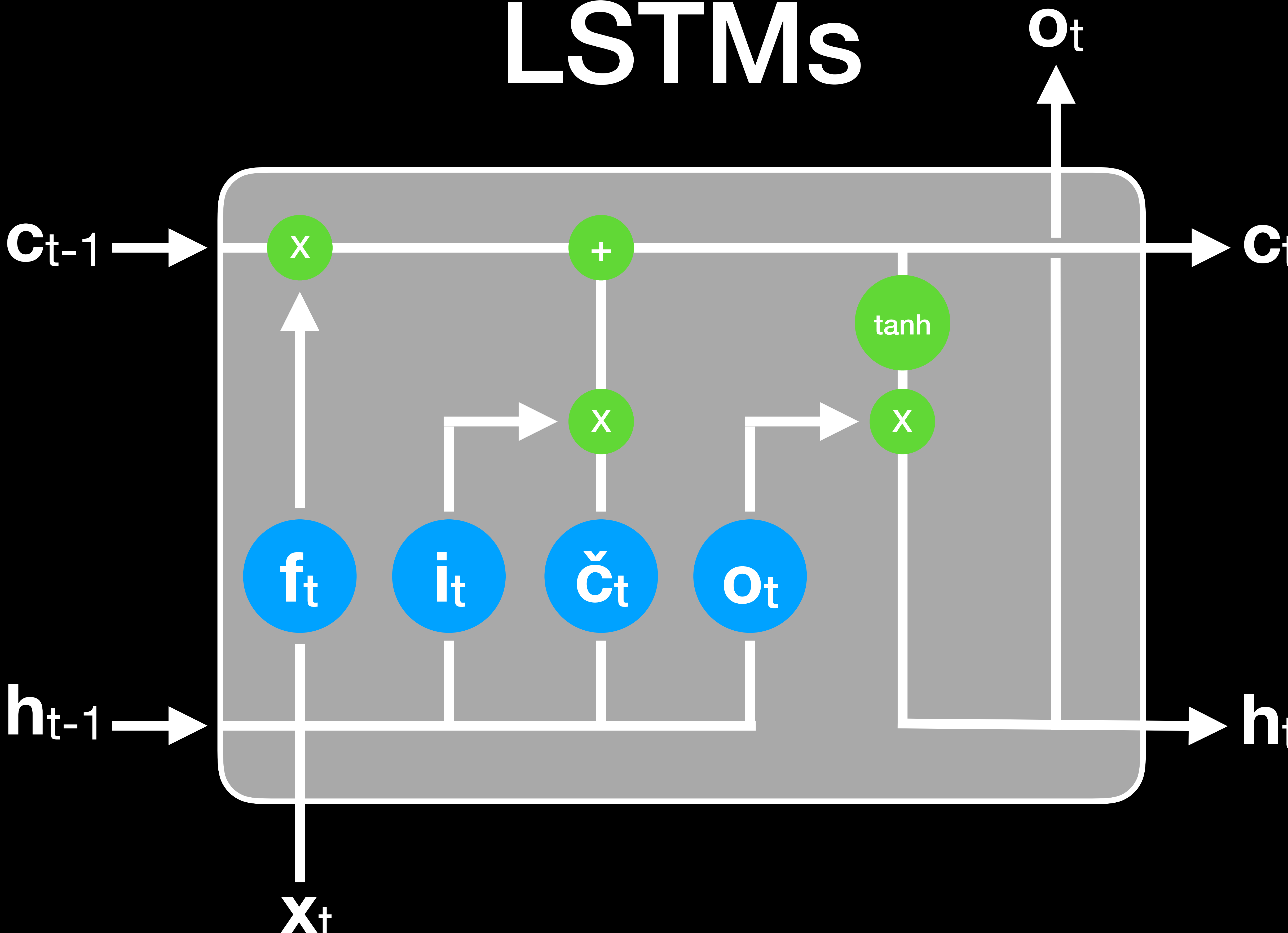
$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$



$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$



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.

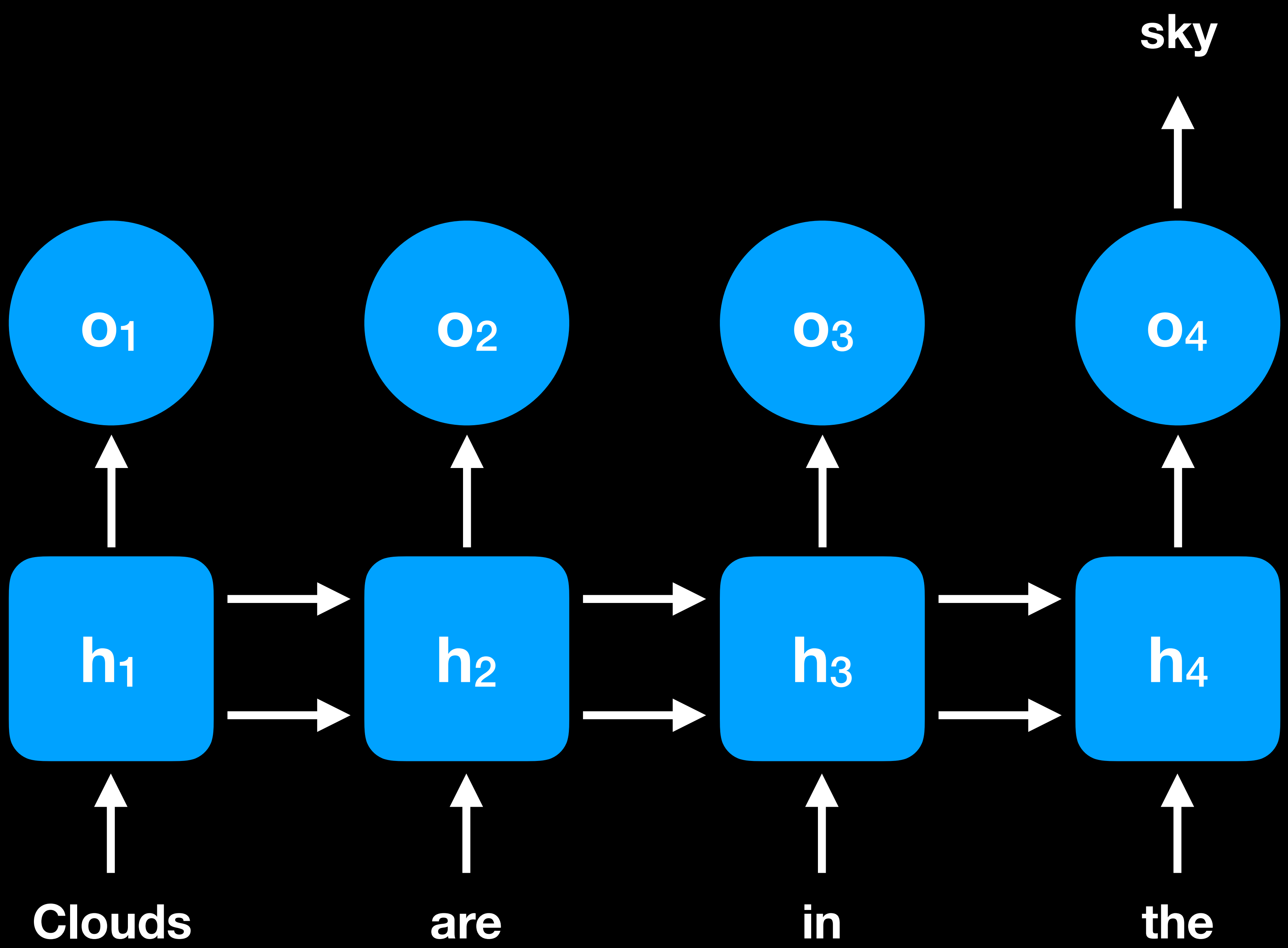
In the LSTM, you're still reusing the same matrices over and over, but they're not directly applied to their output at a previous step: everything is intermediated by the cell state, reducing the potential for vanishing/ exploding gradients and mitigating their impact if they occur[1].

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

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

1. **Because old input can be written out of the cell state afresh at later points in time almost as if were being reinputed, it can affect the gradient w.r.t. error at later points in time even if the original part of the gradient relating to that input vanishes. Techniques like gradient clipping can be used to deal exploding gradients if they occur.**

LSTMs



```python
class RNNTagger(nn.Module):

    def __init__(self, embedding_dim, hidden_dim, vocab_size, tagset_size):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.rnn = nn.RNN(embedding_dim, hidden_dim)
        self.hidden_to_tag = nn.Linear(hidden_dim, tagset_size)
        self.hidden = self.init_hidden()

    def init_hidden(self):
        return torch.zeros(1, 1, self.hidden_dim)

    def forward(self, sentence):
        embeds = self.word_embeddings(sentence)
        reshaped_embeds = embeds.view(len(sentence), 1, -1)
        rnn_out, self.hidden = self.rnn(reshaped_embeds, self.hidden)

        reshaped_rnn_out = rnn_out.view(len(sentence), -1)
        tag_space = self.hidden_to_tag(reshaped_rnn_out)
        tag_scores = F.log_softmax(tag_space, dim=1)

        return tag_scores
```

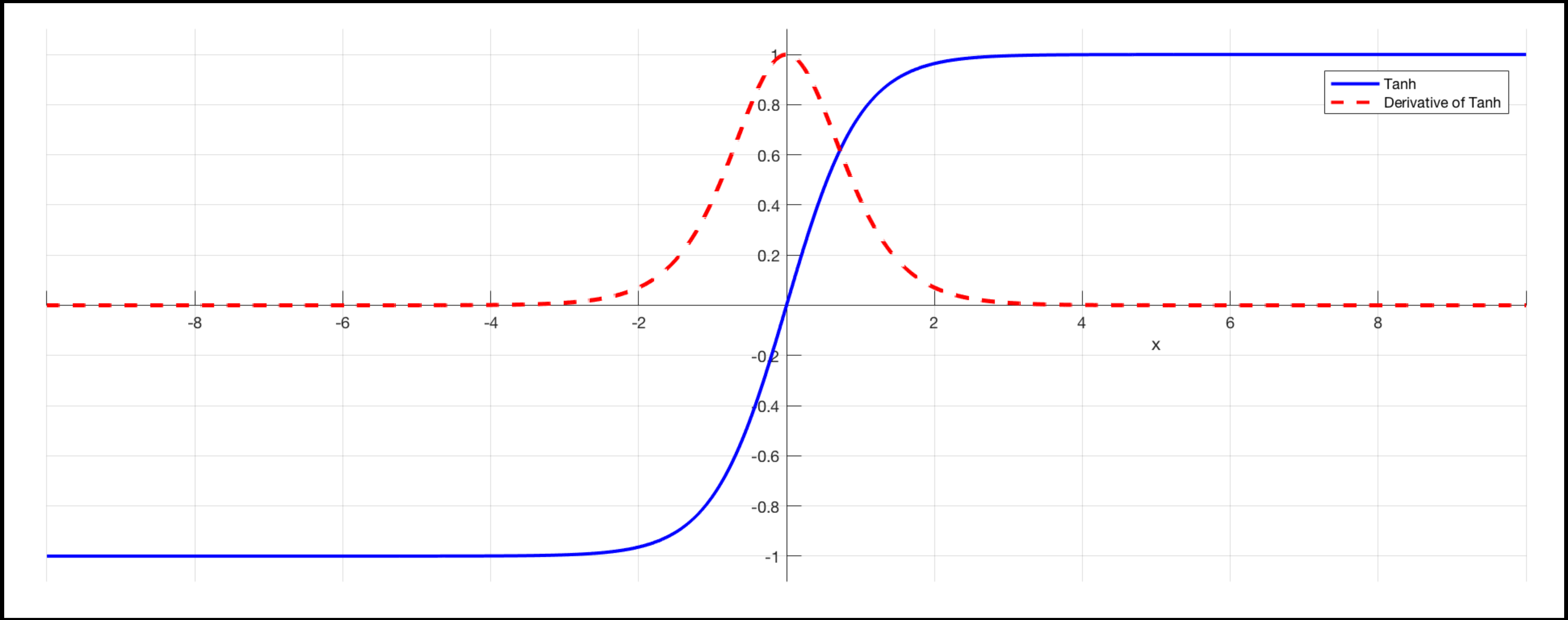
```python
class RNNTagger(nn.Module):

    def __init__(self, embedding_dim, hidden_dim, vocab_size, tagset_size):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.rnn = nn.RNN(embedding_dim, hidden_dim)
        self.hidden_to_tag = nn.Linear(hidden_dim, tagset_size)
        self.hidden = self.init_hidden()

    def init_hidden(self):
        return torch.zeros(1, 1, self.hidden_dim)

    def forward(self, sentence):
        embeds = self.word_embeddings(sentence)
        reshaped_embeds = embeds.view(len(sentence), 1, -1)
        rnn_out, self.hidden = self.rnn(reshaped_embeds, self.hidden)

        reshaped_rnn_out = rnn_out.view(len(sentence), -1)
        tag_space = self.hidden_to_tag(reshaped_rnn_out)
        tag_scores = F.log_softmax(tag_space, dim=1)

        return tag_scores
```

```python
class LSTMTagger(nn.Module):

    def __init__(self, embedding_dim, hidden_dim, vocab_size, tagset_size):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim)
        self.hidden_to_tag = nn.Linear(hidden_dim, tagset_size)
        self.hidden = self.init_hidden()

    def init_hidden(self):
        return (torch.zeros(1, 1, self.hidden_dim),
                torch.zeros(1, 1, self.hidden_dim))

    def forward(self, sentence):
        embeds = self.word_embeddings(sentence)
        reshaped_embeds = embeds.view(len(sentence), 1, -1)
        rnn_out, self.hidden = self.lstm(reshaped_embeds, self.hidden)

        reshaped_lstm_out = rnn_out.view(len(sentence), -1)
        tag_space = self.hidden_to_tag(reshaped_lstm_out)
        tag_scores = F.log_softmax(tag_space, dim=1)

        return tag_scores
```

References

- *Understanding LSTM Networks* by colah
- *Calculus on Computational Graphs: Backpropagation* by colah
- *Deep Learning* by Goodfellow, Bengio, and Courville Pages 205-207, 288-290, 384-388
- *On the difficulty of training Recurrent Neural Networks* by Pascanu, Mikolov, and Bengio

tanh(x)



sigmoid σ(x)