# Recurrent neural networks for neuroscience (Methods)

*Data Science and AI for Neuroscience Summer School, Caltech, July 15, 2022*

Jonathan Kao

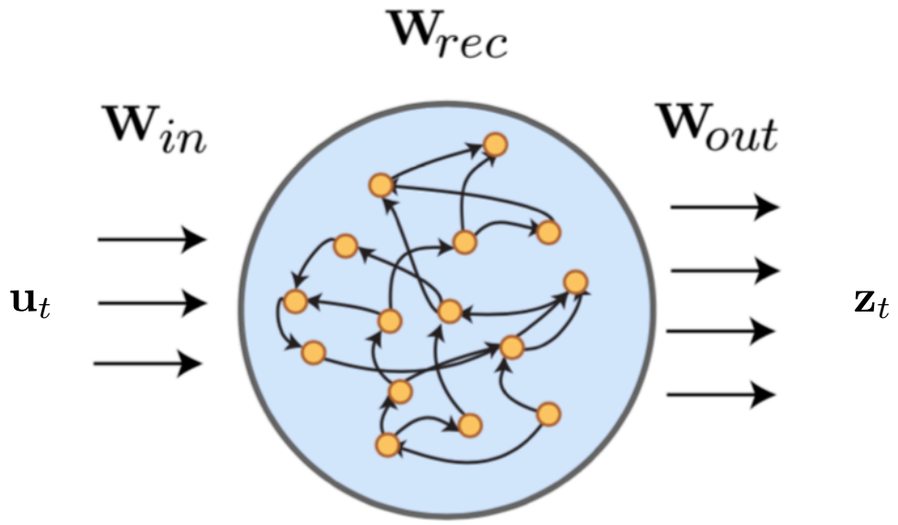
Assistant Professor
Dept of Electrical & Computer Engineering
University of California, Los Angeles

# The continuous time RNN

Params : $W_{in} \in \mathbb{R}^{100 \times 4}$

$W_{rec} \in \mathbb{R}^{100 \times 100}$

$W_{out} \in \mathbb{R}^{1 \times 100}$




$$\tau \dot{\mathbf{x}}_t = -\mathbf{x}_t + \mathbf{W}_{\text{rec}} f(\mathbf{x}_t) + \mathbf{W}_{\text{in}} \mathbf{u}_t + \mathbf{b}$$

$$\mathbf{h}_t = f(\mathbf{x}_t)$$

$$\mathbf{z}_t = \mathbf{W}_{\text{out}} f(\mathbf{x}_t)$$

$z_t \in \mathbb{R}^1$

The RNN has three major components:

- $\mathbf{W}_{\text{in}}$: An input at time $t$, denoted $\mathbf{u}_t$ is transformed via $\mathbf{W}_{\text{in}}$ onto artificial neurons, whose activations are $\mathbf{h}_t$.
- $\mathbf{W}_{\text{rec}}$: Each artificial neuron is the network is denoted by an orange circle, and these artificial neurons have recurrent connections. recurrent connections are defined by the matrix $\mathbf{W}_{\text{rec}}$.
- $\mathbf{W}_{\text{out}}$: Finally, the artificial neuron activations are mapped linearly to the output $\mathbf{z}_t$ through the matrix $\mathbf{W}_{\text{out}}$.

$u_t$ : RNN input @ time $t$. $\in \mathbb{R}^4$

$x_t$ : pre-activation activity $\in \mathbb{R}^{100}$

$h_t \in \mathbb{R}^{100}$ : hidden state of neuron

$h_t = f(x_t)$

$\tau \dot{x}_t = -x_t$

# Euler expansion and deep learning vanilla RNN

$$\tau \dot{\mathbf{x}}_t = -\mathbf{x}_t + \mathbf{W}_{\text{rec}} f(\mathbf{x}_t) + \mathbf{W}_{\text{in}} \mathbf{u}_t + \mathbf{b}$$

$$\mathbf{z}_t = \mathbf{W}_{\text{out}} f(\mathbf{x}_t)$$

Vanilla:

$$h_{t+1} = f(W_{rec} h_t + W_{in} u_t + b)$$

$$\dot{x}_t = \frac{dx}{dt} = \frac{x_{t+\Delta t} - x_t}{\Delta t}$$

$$\alpha = \frac{\Delta t}{\tau}$$

$$\Rightarrow \quad \tau \cdot \frac{x_{t+1} - x_t}{\Delta t} = -x_t + W_{rec} f(x_t) + W_{in} u_t + b$$

$$x_{t+1} - x_t = \alpha \cdot (\quad)$$

$$x_{t+1} = (1 - \alpha) x_t + \alpha (W_{rec} f(x_t) + W_{in} u_t + b)$$

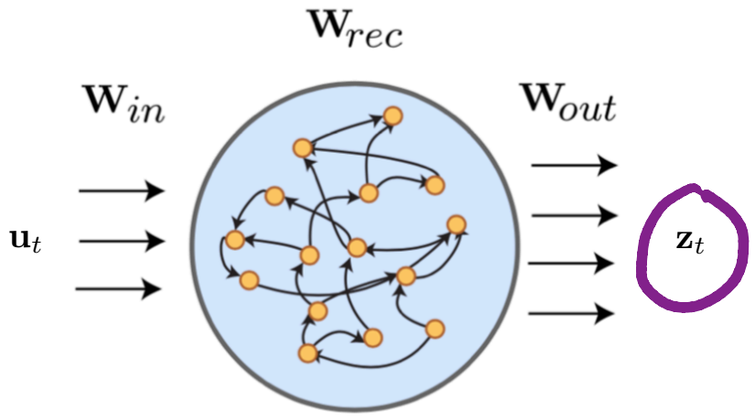
$$\alpha = 1$$

$$f(x_{t+1}) = f\left(W_{rec}\, f(x_t) + W_{in}\, u_t + b\right)$$

$$\Downarrow$$

$$h_{t+1} = f\left(W_{rec}\, h_t + W_{in}\, u_t + b\right)$$

# How is an RNN trained?




$$\dot{\mathbf{x}}_t = -\mathbf{x}_t + \mathbf{W}_{\text{rec}} f(\mathbf{x}_t) + \mathbf{W}_{\text{in}} \mathbf{u}_t + \mathbf{b}_{\text{rec}}$$

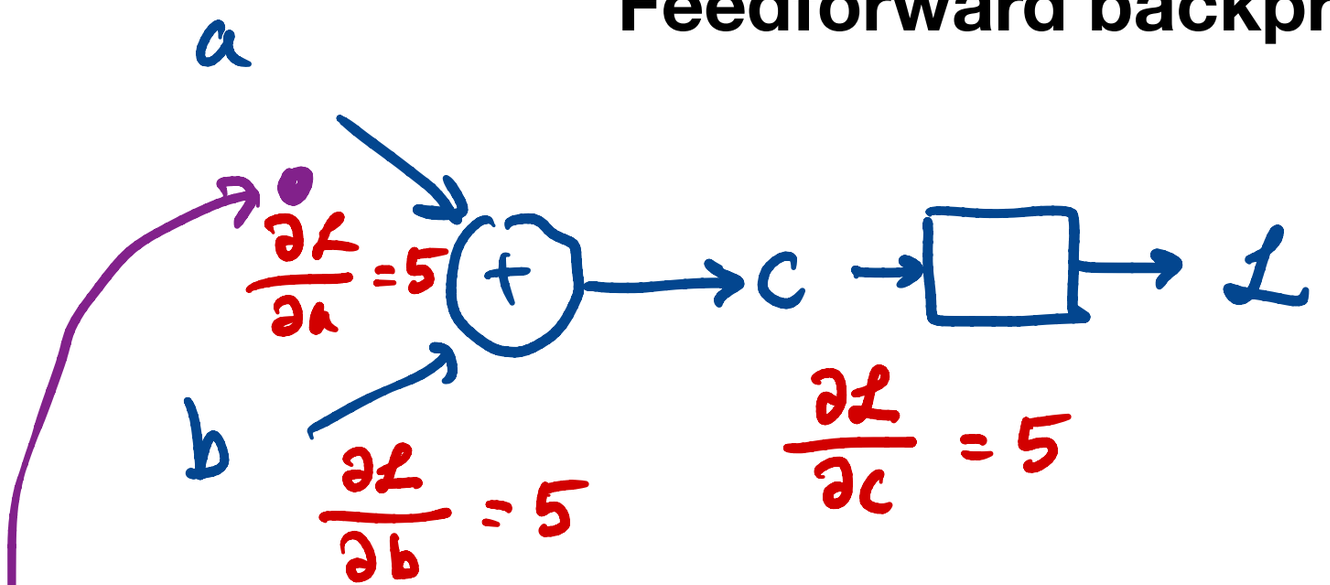
```
########
## TODO:
##    Do the following:
##      1. Compute the output and loss over the inputs using train_one_batch().
##      2. Clear the gradients in PyTorch using optimizer.zero_grad()
##      3. Compute the gradients using loss.backward()
##      4. Take an optimizer step using optimizer.step()
########
```

We need:

- To know how good our model is doing (**loss function**). $\mathcal{L}(y_t, z_t) = \frac{1}{T} \sum_{t=1}^{T} \| z_t - y_t \|^2$
- How to modify the weights to do better (**gradients with respect to the loss function**).
- An optimizer to do gradient descent (**beyond scope, we will use Adam**).

$$W_{rec} \leftarrow W_{rec} - \varepsilon \nabla_{W_{rec}} \mathcal{L}$$

# Feedforward backpropagation



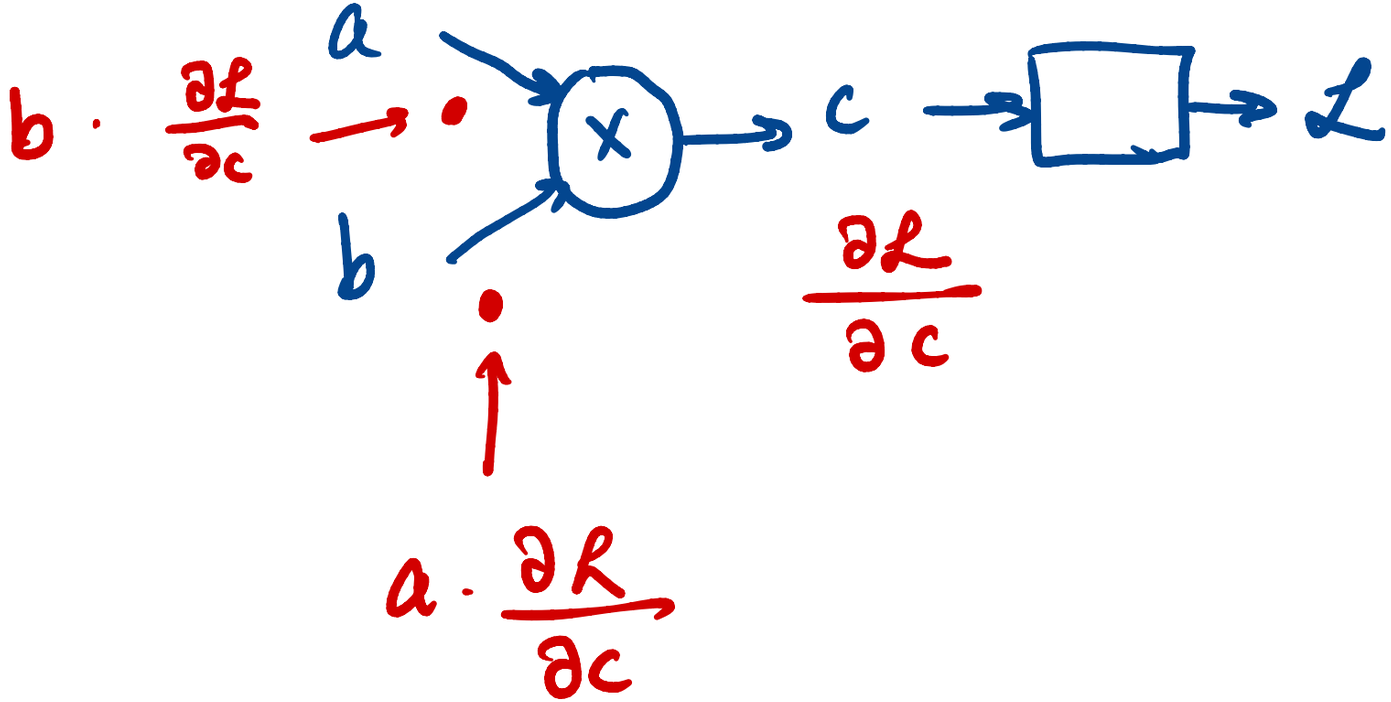

$$c = a + b$$

$$\frac{\partial c}{\partial a} = 1 \,, \quad \frac{\partial c}{\partial b} = 1$$

$$\frac{\partial L}{\partial a} = \frac{\partial c}{\partial a} \cdot \frac{\partial L}{\partial c}$$

$$\Delta c \approx \frac{\partial c}{\partial a} \Delta a$$

# Feedforward backpropagation




$$c = a \cdot b$$

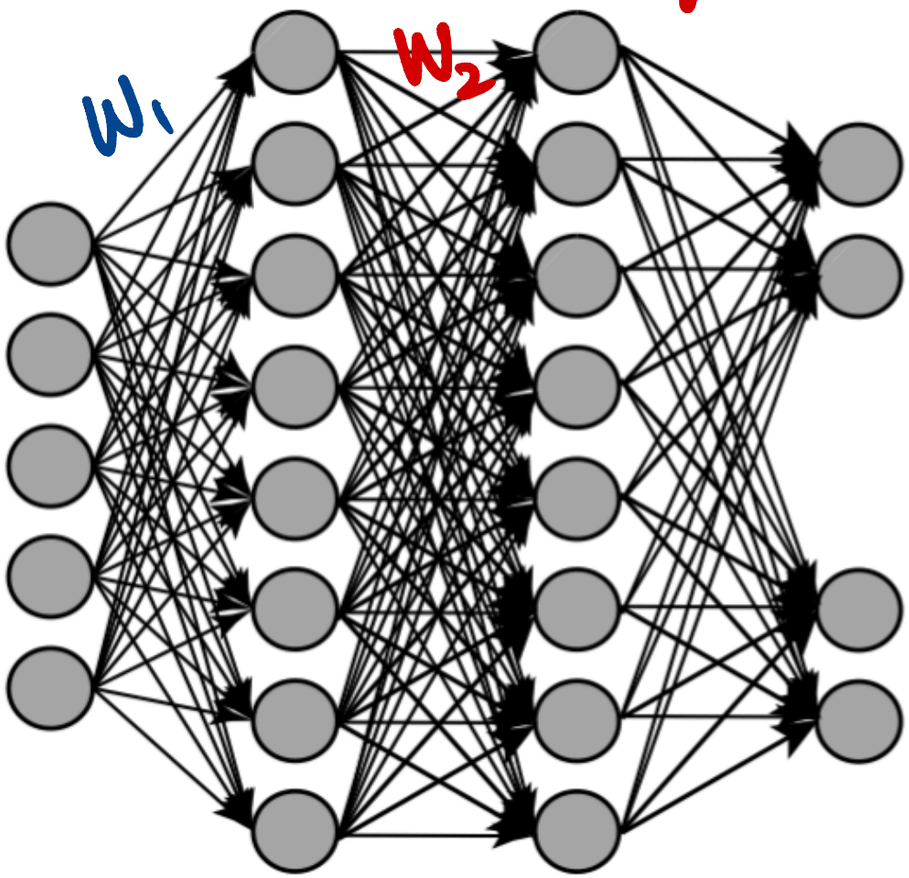
$$\frac{\partial c}{\partial a} = b$$

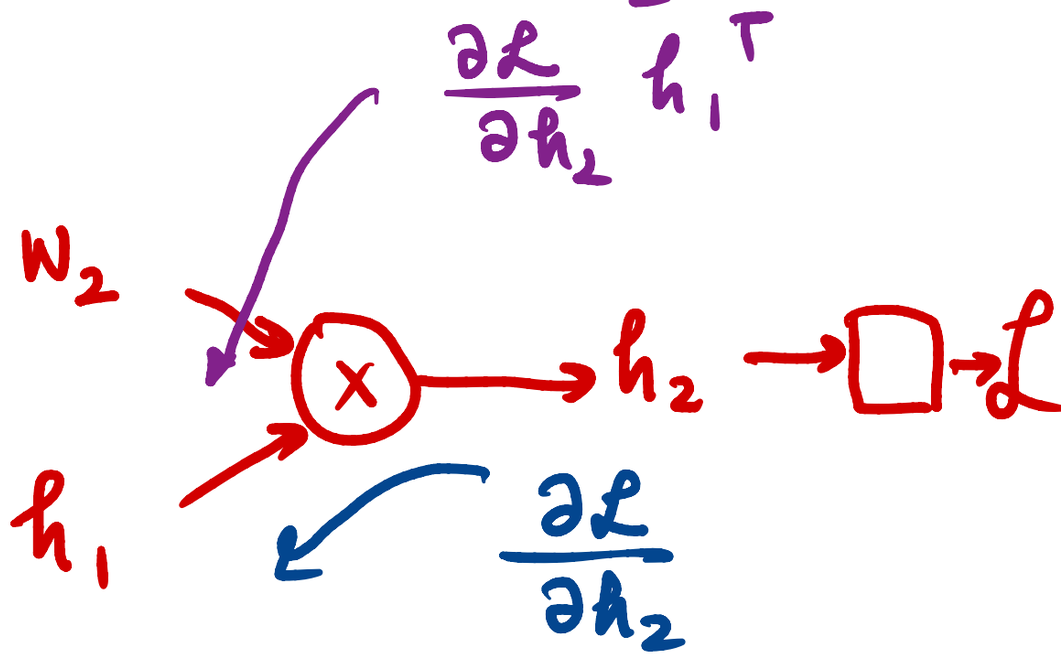
$$\frac{\partial c}{\partial b} = a$$

# Feedforward backpropagation

$h_2 = f(x_2) \Rightarrow \frac{\partial h_2}{\partial x_2}$

$h_2 = f(W_2 h_1)$

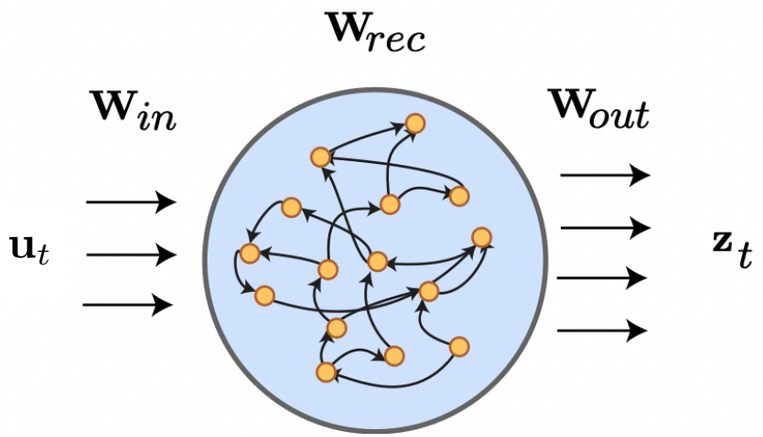



$W_2^T \frac{\partial \ell}{\partial h_2}$

# How is an RNN trained?

Training an RNN is not immediately as straightforward as a feedforward neural network. This is because the RNN has recurrent connections with loops, and backpropagation is not straightforward.

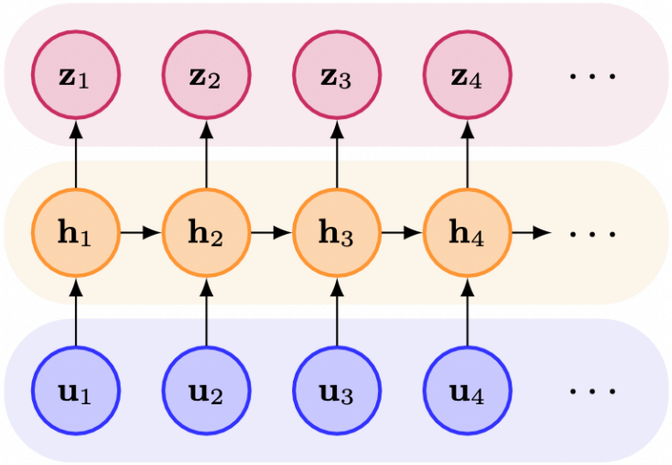



The upstream gradients at any given time come from units who are themselves potentially receiving inputs (directly or indirectly) from the node we're trying to calculate the gradient of. Further, the activations at any given node for an input depends on time; for even an input $\mathbf{u}_t$ that is static across time, the activations $\mathbf{h}_t$ will not be static across time.

# How is an RNN trained?

$$h_{t+1} = f(W_{rec} h_t + W_{in} u_t + b)$$

To get around this confound, we consider the RNN as a computational graph through time.




Importantly, each of the $\mathbf{h}_t$ are the activations of all the hidden units at a given time. If we expand the computational graph through time, then we effectively see that its a feedforward neural network, where the number of layers is the number of time steps.

It is for this reason that Schmidhuber views RNNs as the deepest of neural networks (Schmidhuber, 2016), since they are effectively feedforward networks with depth given by time.

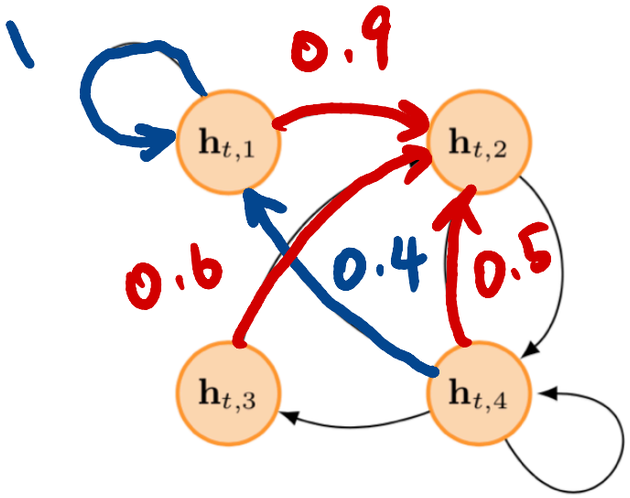
$$h_{t+1} = f(W_{rec} h_t)$$

# How is an RNN trained?

For example, consider a matrix $\mathbf{W}_{\text{rec}}$ that looks like the following:

$$\begin{bmatrix} h_{t+1,1} \\ h_{t+1,2} \\ \vdots \\ h_{t+1,4} \end{bmatrix} = f\left( \begin{bmatrix} 1 & 0 & 0 & 0.4 \\ 0.9 & 0 & 0.6 & 0.5 \\ 0 & 0 & 0 & 0.3 \\ 0 & 0.8 & 0 & 0.7 \end{bmatrix} \begin{bmatrix} h_{t,1} \\ h_{t,2} \\ h_{t,3} \\ h_{t,4} \end{bmatrix} \right)$$

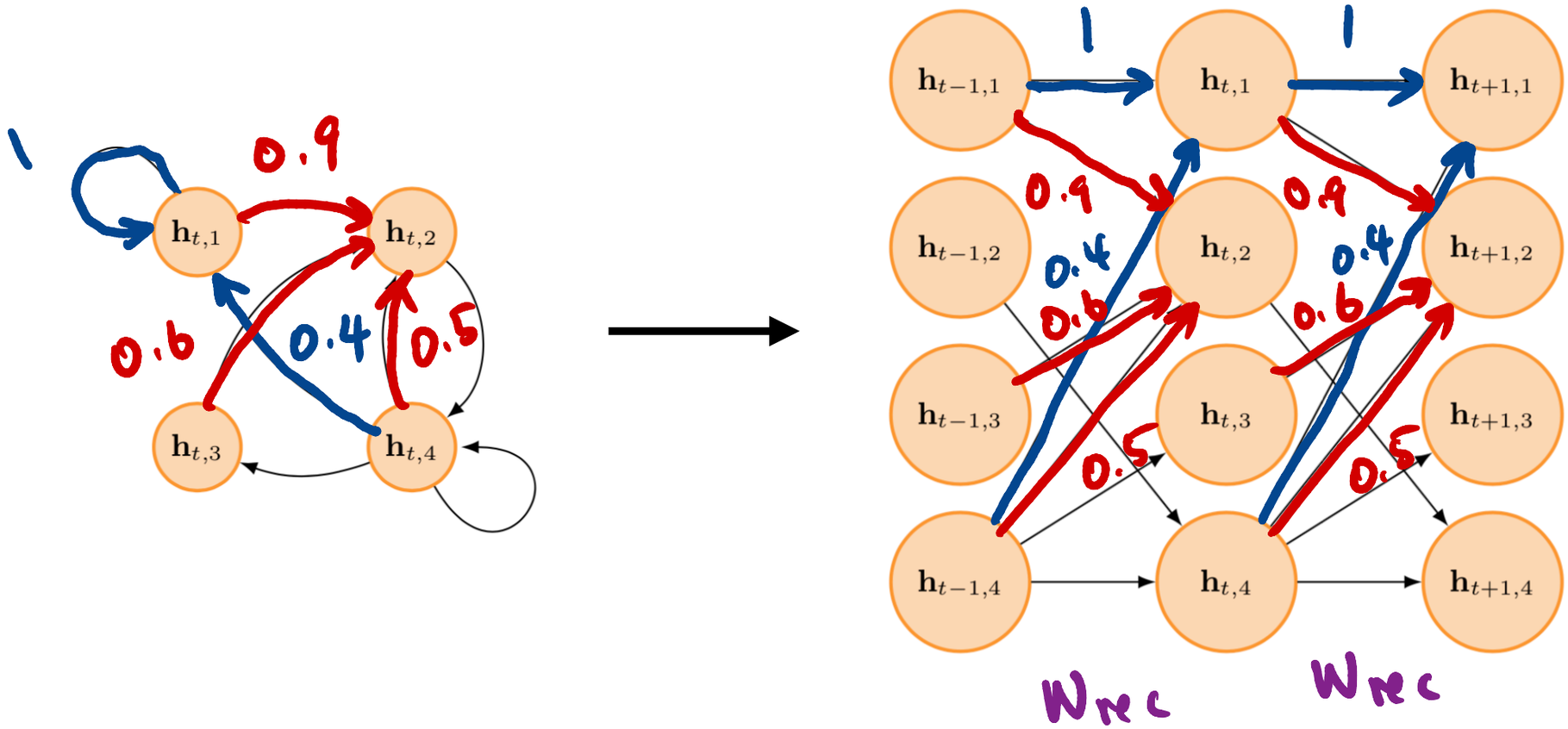
This RNN looks like the following.



# How is an RNN trained?

MRNN

$W_{rec} = A \, \mathrm{diag}(u_t) \, B$

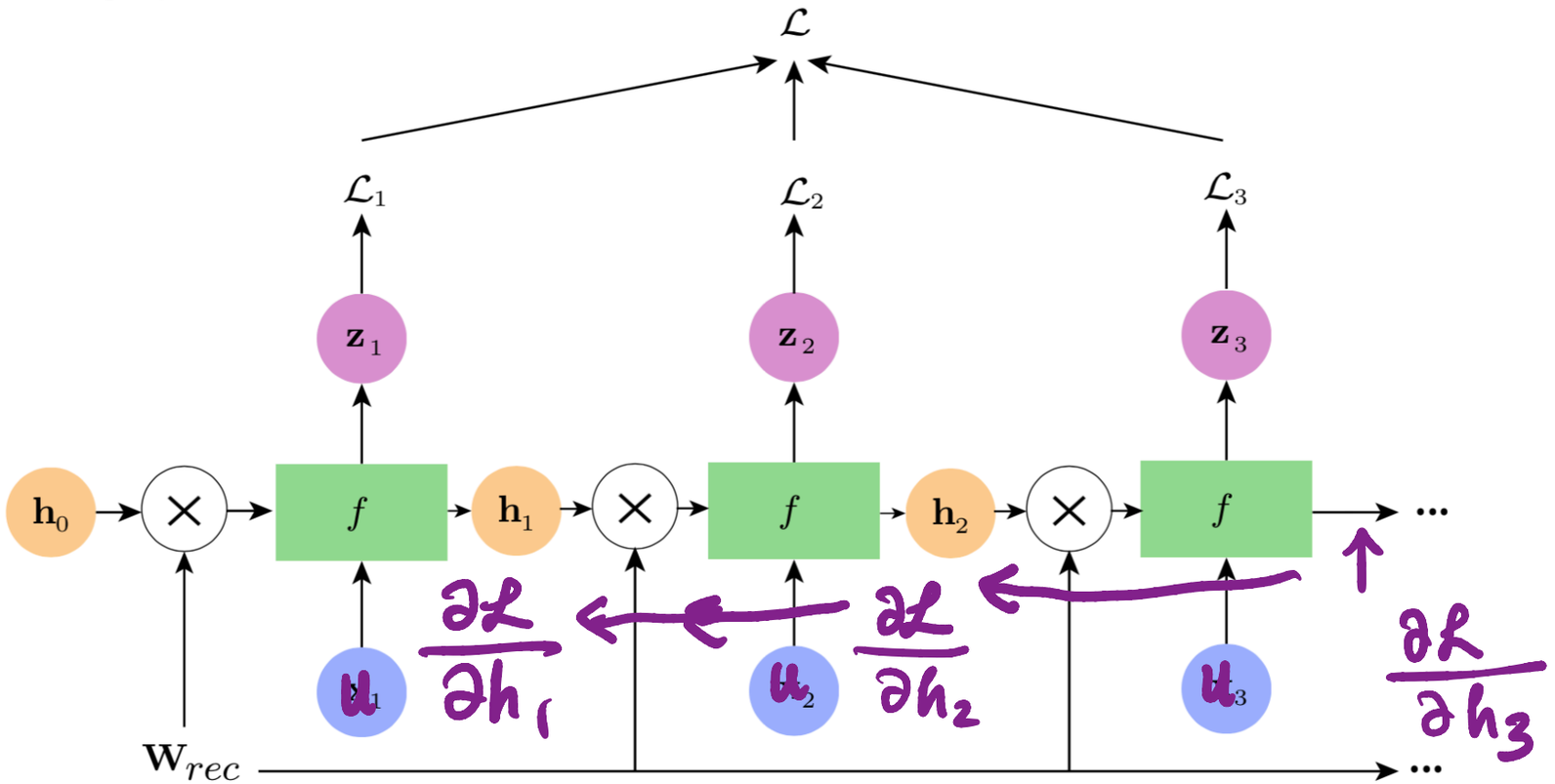


**We do this in PyTorch (Brandon will walk you through in more detail later)**

# A practical challenge of training RNNs

## RNN training (cont.)

This graph can be redrawn as follows:




**Bottom line:** the computational graph has repeated multiplications by $\mathbf{W}_{\text{rec}}$.

# A practical challenge of training RNNs

**Why does repeated multiplication by $\mathrm{W}_{\mathrm{rec}}$ make training more difficult?**

$$W_{rec}^{t} = (U \Lambda U^{-1})^{t}$$

$$= U \Lambda U^{-1} U \Lambda U^{-1} \cdots U \Lambda U^{-1}$$

$$= U \Lambda^{t} U^{-1}$$

$$\Lambda^{t} = \begin{bmatrix} \lambda_1^{t} & & \\ & \lambda_2^{t} & \\ & & \ddots \\ & & & \lambda_N^{t} \end{bmatrix}$$

$\lambda_1 > 1$

$\lambda_1 = 1.1 \Rightarrow 1.1^{100}$

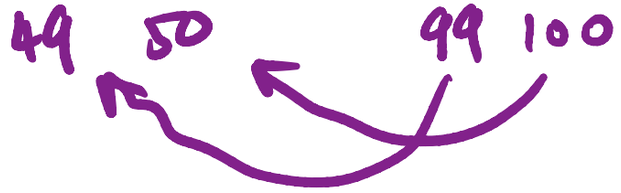
$= 13780$

$\lambda_1 = 0.9 \Rightarrow 0.9^{100}$

$= 2.7 \times 10^{-5}$

# Addressing vanishing and exploding gradients

**Methods of addressing:**




- In general, we can truncate BPTT.
- For exploding gradients, we gradient clip.

$$\|g\| > c \longrightarrow g = \frac{c}{\|g\|} g$$

- For vanishing gradients, we could employ regularizations (Pascanu et al., 2012).

$$\Omega = \sum_t \left( \frac{\left\| \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t} \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}} \right\|}{\left\| \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}} \right\|} - 1 \right)^2$$

- Weight initialization strategies.

$$W_{rec} = I$$

- In machine learning applications, vanilla RNNs are rarely used. Most use LSTMs or other architectures.

# Long short-term memory (LSTM)

## The long short-term memory

The long short-term memory (LSTM) is a particular RNN architecture that is well-suited for addressing the problem of vanishing and exploding gradients. It was proposed by Hochreiter and Schmidhuber in 1997. It is one of the most commonly used RNN architectures today.

The name refers to its ability to store *short-term* memory for a *long* period of time. Hopefully the next few slides will help unpack what this really means.

# Long short-term memory (LSTM)

## LSTM

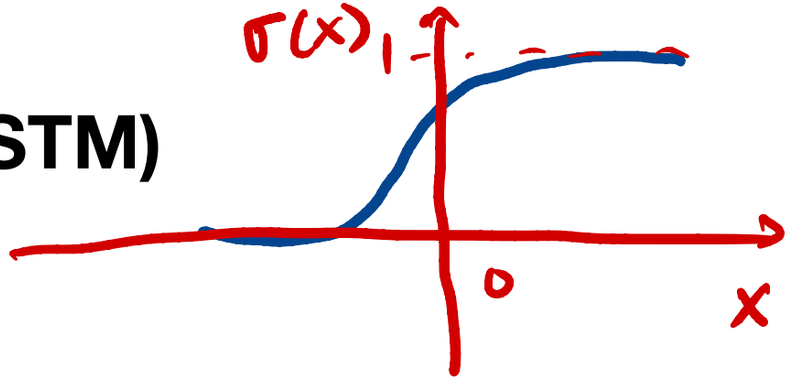
The standard LSTM is defined as follows:

$$
\begin{aligned}
\mathbf{f}_t &= \sigma\left(\mathbf{W}_f \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_f\right) \\
\mathbf{i}_t &= \sigma\left(\mathbf{W}_i \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_i\right) \\
\mathbf{v}_t &= \tanh\left(\mathbf{W}_v \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_v\right) \\
\mathbf{o}_t &= \sigma\left(\mathbf{W}_o \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_o\right) \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{v}_t \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t)
\end{aligned}
$$

where $\sigma(\cdot)$ is the sigmoid function. These functions at first glance are opaque. Here, we will unpack what they mean and how they help solve the vanishing and exploding gradients problem.

$u_t \Rightarrow x_t$

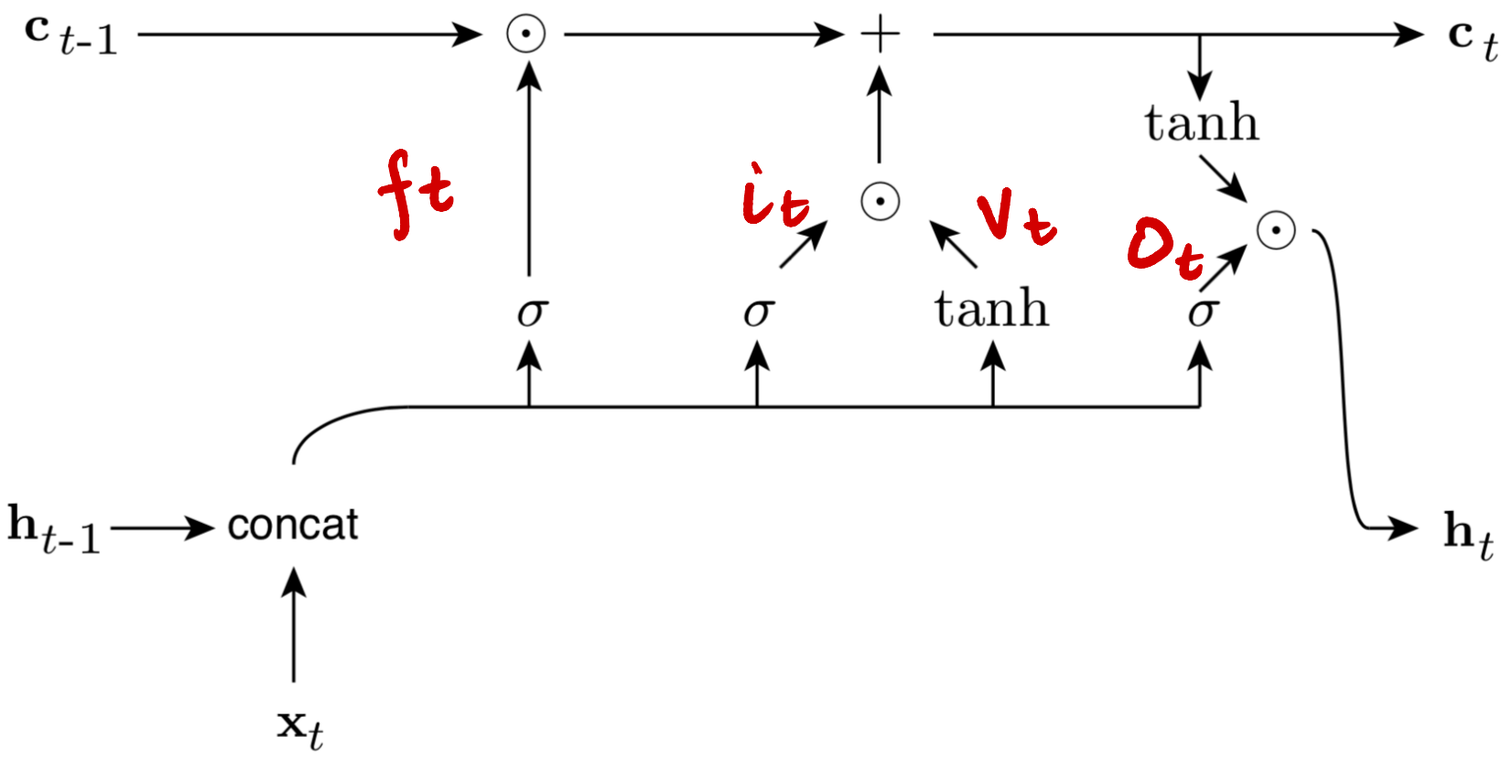
# Long short-term memory (LSTM)




## LSTM, block diagram

Here is a block diagram of the LSTM cell.

$$
\begin{aligned}
\mathbf{f}_t &= \sigma\left(\mathbf{W}_f \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_f\right) \\
\mathbf{i}_t &= \sigma\left(\mathbf{W}_i \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_i\right) \\
\mathbf{v}_t &= \tanh\left(\mathbf{W}_v \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_v\right) \\
\mathbf{o}_t &= \sigma\left(\mathbf{W}_o \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_o\right) \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{v}_t \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t)
\end{aligned}
$$



# Long short-term memory (LSTM)

## LSTM, cell state

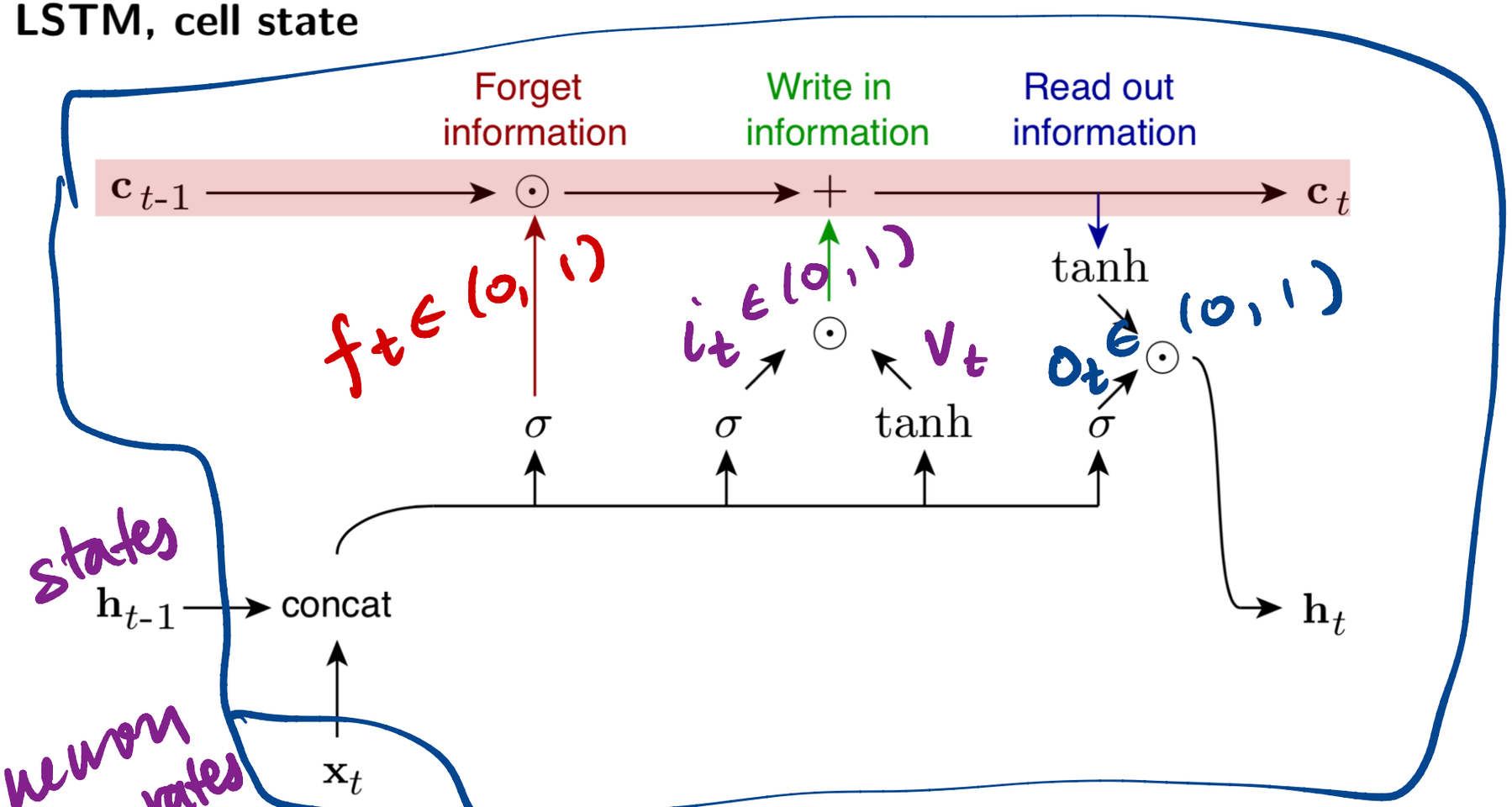
The LSTM cell state, $\mathbf{c}_t$, is the central component of the LSTM. We think of the cell state as some memory or tape; it holds some value and remembers it. But the key thing is that we can alter this cell state (i.e., we can alter the memory or tape). At each point in time, there are three things we can do to the cell state:

1. Forget information.
2. Write-in information.
3. Read-out information.

This is illustrated on the next page.

Note that the next hidden state, $\mathbf{h}_t$, is essentially a read out of the cell state $\mathbf{c}_t$, as $\mathbf{h}_t = f(\mathbf{c}_t)$, so the cell state contains all the vital information in the LSTM.

# Long short-term memory (LSTM)




Forgetting information, writing in information, and reading out information are all mediated by gates.

# Long short-term memory (LSTM)

## LSTM, forgetting information

To forget information, the LSTM uses the forget gate, $\mathbf{f}_t$. This comprises the update, $\mathbf{c}_t = \mathbf{c}_{t-1} \odot \mathbf{f}_t$. Note:

- If $\mathbf{f}_t$ is close to $1$, then $\mathbf{c}_t \approx \mathbf{c}_{t-1}$. Thus, when the forget gate is large, most information is remembered.
- If $\mathbf{f}_t$ is close to $0$, then $\mathbf{c}_t \approx 0$. Thus, when the forget gate is small, most information is forgotten.

This gate is illustrated on the next page.

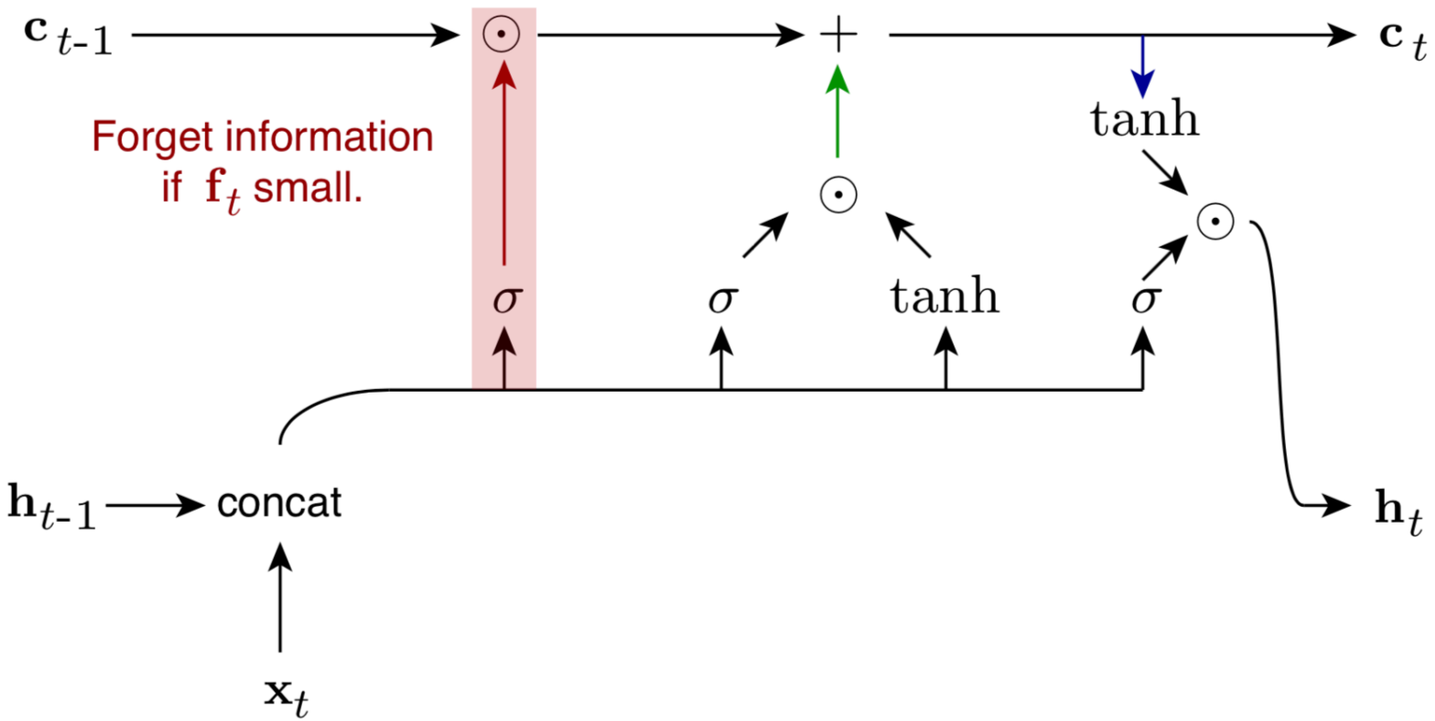
# Long short-term memory (LSTM)

## LSTM, forget gate

The network computes a linear transformation

$$\mathbf{f}_t = \sigma\left(\mathbf{W}_f \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_f\right)$$

and if $\mathbf{f}_t$ is small, it forgets the information. If $\mathbf{f}_t$ is large, it remembers the information.



# Long short-term memory (LSTM)

## LSTM, writing in information

To write in information, the LSTM needs to compute two values. The following is not convention, but it helps me remember. We'll call these gates the value gate, $\mathbf{v}_t$, and the input gate, $\mathbf{i}_t$. These are calculated as:

$$\begin{aligned}\mathbf{v}_t &= \tanh\left(\mathbf{W}_v \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_v\right) \\ \mathbf{i}_t &= \sigma\left(\mathbf{W}_i \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_i\right)\end{aligned}$$

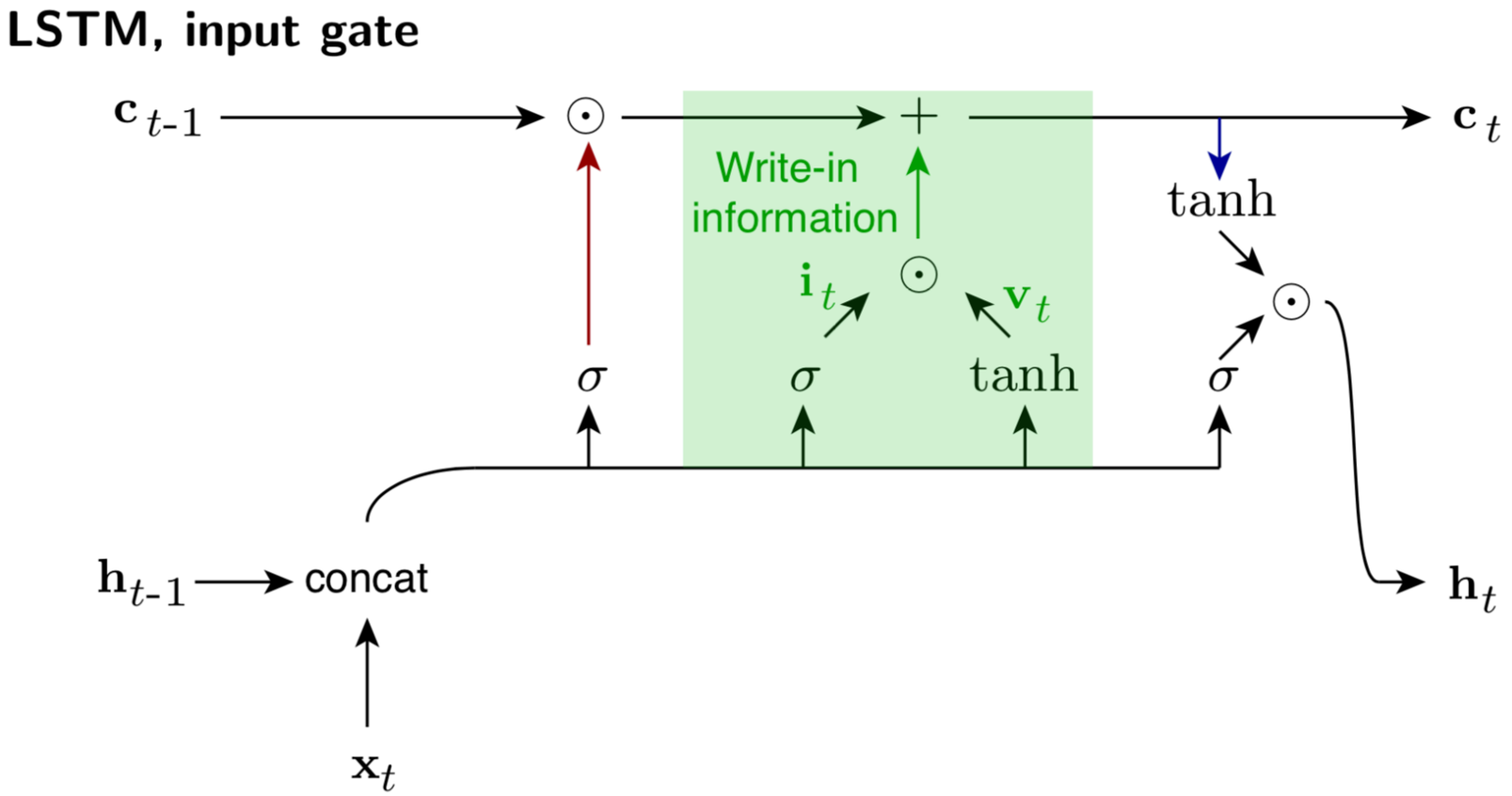
- The value gate, $\mathbf{v}_t \in (-1, 1)$ tells us the value we want to add to the cell state.
- The input gate, $\mathbf{i}_t \in (0, 1)$ tells us how much of the value gate, $\mathbf{v}_t$ to write to the cell state.

Then, the total information we get to write in to the cell state is the multiplication of these two, i.e., this update looks like:

$$\mathbf{c}_t = \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{v}_t$$

If either $\mathbf{v}_t$ or $\mathbf{i}_t$ is zero, we write no information to the cell state. If $\mathbf{i}_t = 1$, we write whatever update information $\mathbf{v}_t$ into the cell.

# Long short-term memory (LSTM)



# Long short-term memory (LSTM)

## LSTM, reading out information

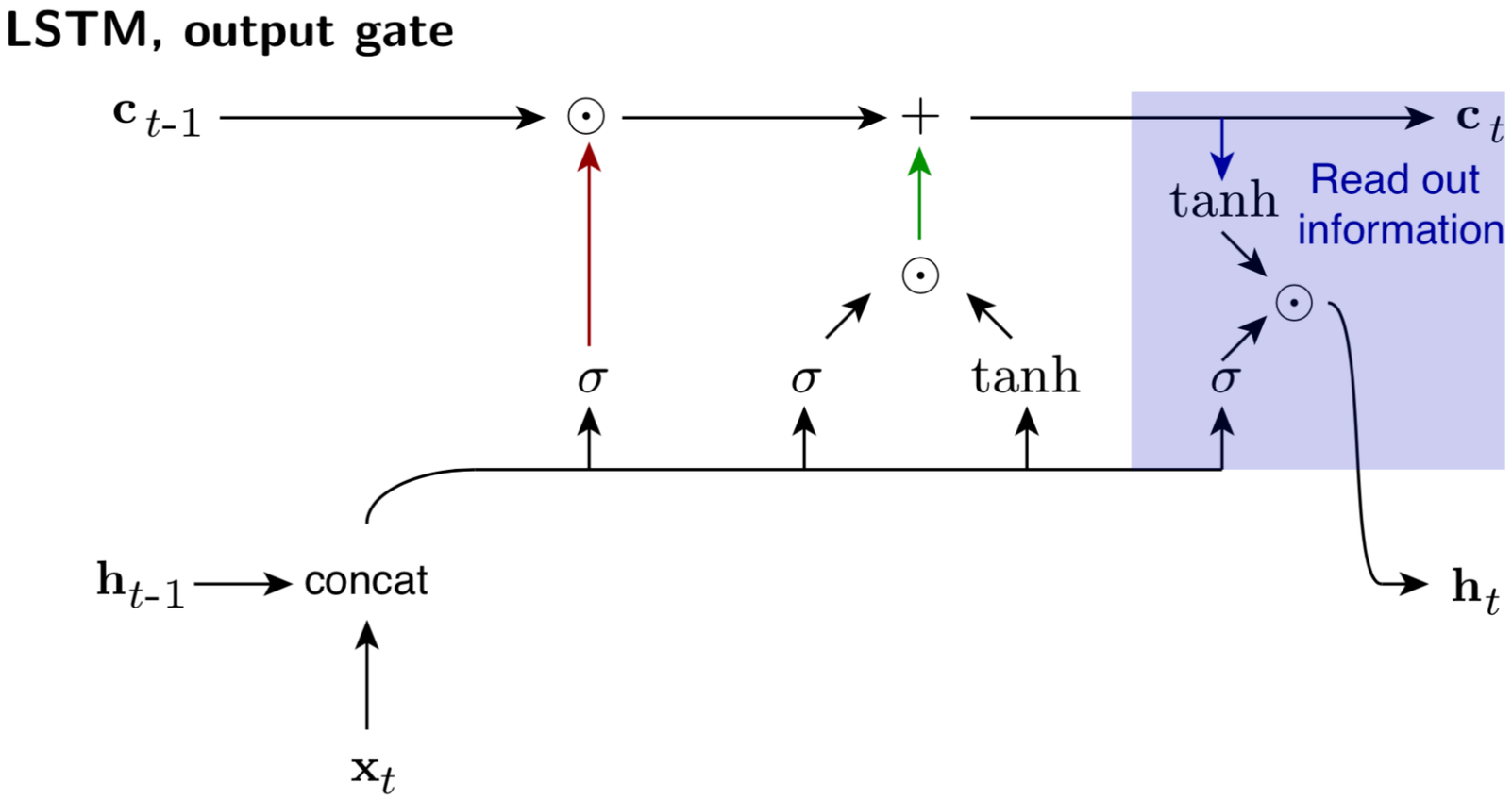
Now that we've updated the cell state by both forgetting information and writing in information, we are now ready to read out information to update the hidden state of the LSTM. This is fairly simple: we take our cell state, put it through the $\tanh$ nonlinearity, and then use our final gate, the output gate $\mathbf{o}_t$.

- The output gate, $\mathbf{o}_t$, tells us how much of the cell state is going to be read out to the hidden state.
- If $\mathbf{o}_t$ is small, very little will be read out.
- If $\mathbf{o}_t$ is large, a lot of the cell state will be read out.

Formally, the readout is:

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

# Long short-term memory (LSTM)



# Long short-term memory (LSTM)

## LSTM training

From this representation of the LSTM, we now can glean insight into why this works well.

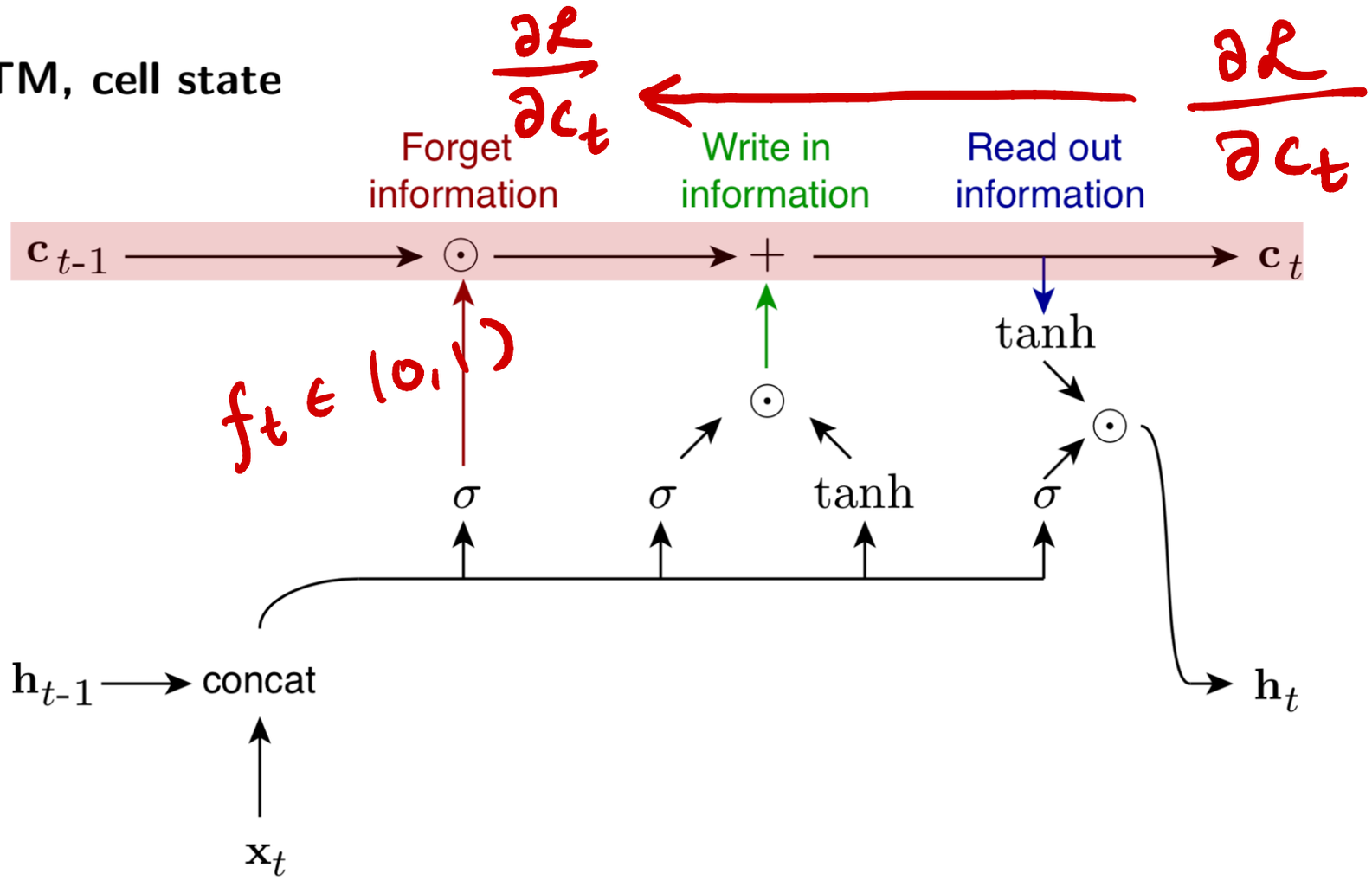
- The cell state, $\mathbf{c}_t$, during backpropagation, has almost uninterrupted gradient flow. It's analogous to a gradient highway, much like in ResNets.
- In particular, the $+$ operation passes the gradient back.
- The gradient may be attenuated by the forget gate, $\mathbf{f}_t$. The gradient

$$\frac{\partial \mathcal{L}}{\partial \mathbf{c}_{t-1}} = \frac{\partial \mathcal{L}}{\partial \mathbf{c}_t} \odot \mathbf{f}_t$$

and therefore if the forget gate is small, then $\frac{\partial \mathcal{L}}{\partial \mathbf{c}_{t-1}}$ will be small, too.

# Long short-term memory (LSTM)

**LSTM, cell state**




Forgetting information, writing in information, and reading out information are all mediated by gates.

# Long short-term memory (LSTM)

## LSTM, last comments

In practice, LSTMs are easier to train than vanilla RNNs using first order gradient descent techniques. However, note that the LSTM has (!) $4\times$ the number of parameters of a vanilla RNN.

To address this problem, another type of recurrent unit is used, called the **gated recurrent unit**.

# Identifying fixed points in an RNN

$$\tau \dot{\mathbf{x}}_t = -\mathbf{x}_t + \mathbf{W}_{\text{rec}} f(\mathbf{x}_t) + \mathbf{W}_{\text{in}} \mathbf{u}_t + \mathbf{b}$$

$\dot{x}_t = 0$

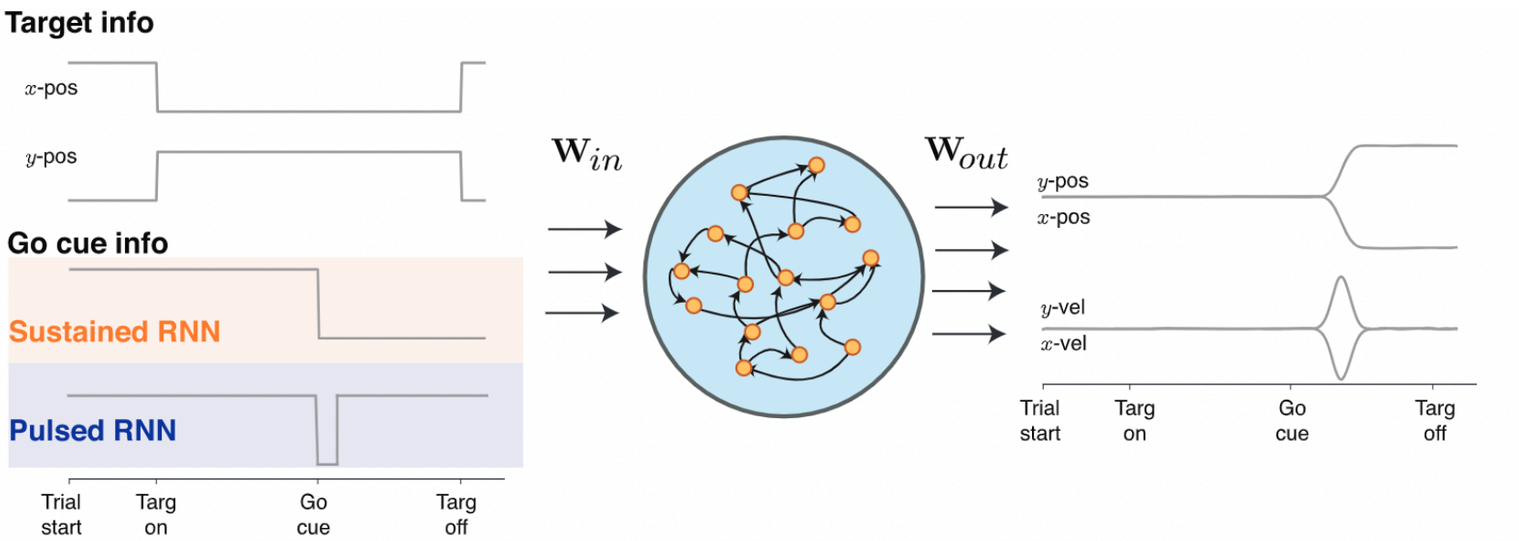
$q(x) = -x + W_{rec} f(x) + W_{in} u + b$

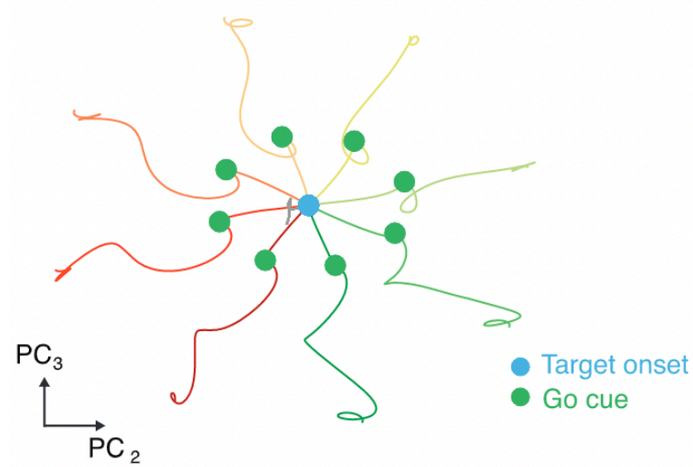
$\min_x \quad \frac{1}{2} \| q(x) \|^2$

Use your fav. optimizer

# Fixed points depend on the input

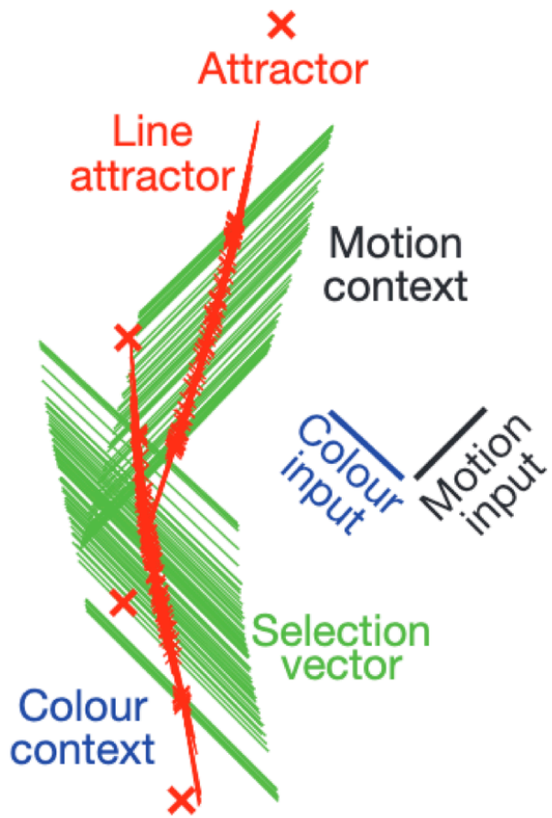
You can solve for them when the input is zero, but sometimes it's also interesting to consider when they are non-zero.



# Slow points

It may sometimes be desirable to identify slow points by setting a threshold on the objective.



# Methods lecture take home points

- We derived the discrete time update of the continuous RNN and related it to the vanilla RNN.
- We discussed how RNNs can be trained via BPTT and the vanishing + exploding gradients problem.
- We talked about the LSTM architecture and how it avoids this.
- We discussed how to find fixed points of RNNs.