# DD2437 – Artificial Neural Networks and Deep Architectures (annda)

## Lecture 8: **Temporal processing with ANNs: feedforward vs recurrent networks**

Pawel Herman

Computational Science and Technology (CST)

KTH Royal Institute of Technology

# Lecture overview

- Temporal processing with feedforward NNs
- Recurrent architectures for sequence modelling
- Backpropagation through time (BPTT)
- Echo state networks (ESNs)
- Long short-term memory model (LSTM)

# Temporal aspects

- Time is an essential component of the description of many phenomena, observations, data structures
- Omnipotence of sequences – ordering of entities
  - numerical codes
  - language and speech
  - motor behaviour
  - signals, time series: sensor readings, market prices, biological recordings etc.
- Discrete vs continuous time
- Implicit vs explicit representation

# Static MLP for handling dynamics

The use of a static MLP to account for temporal dimension

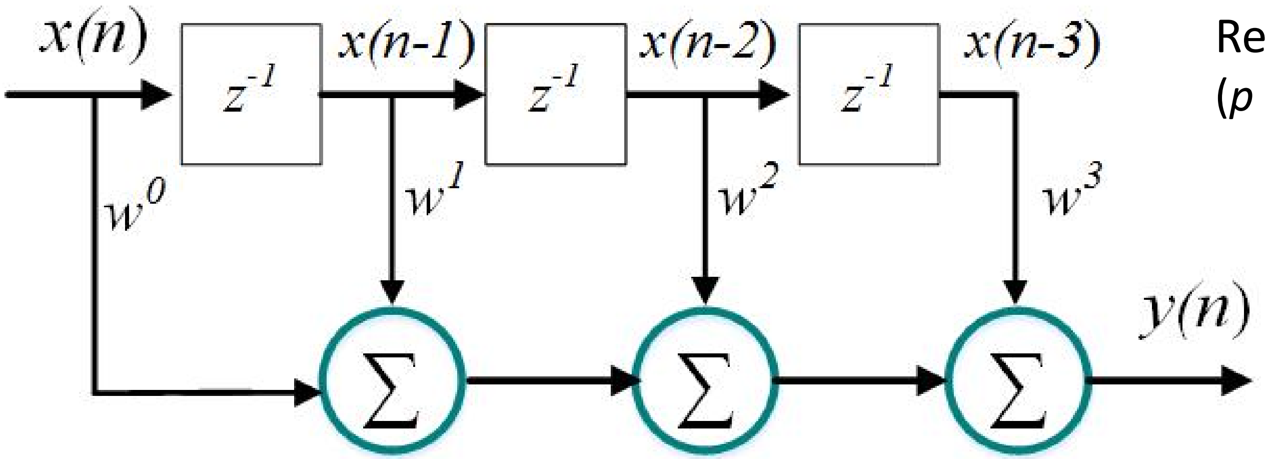
- short-term memory function
- nonlinear regression capabilities

# Static MLP for handling dynamics

The use of a static MLP to account for temporal dimension

- short-term memory function
- nonlinear regression capabilities

Tapped delay line memory



Memory depth = *p* (3)

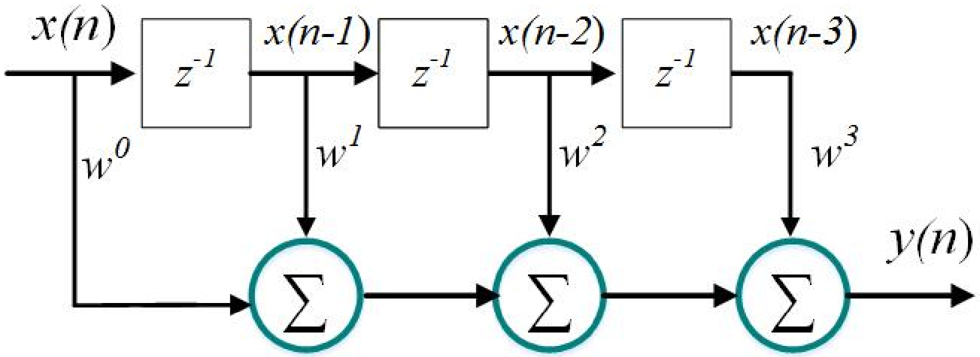
Resolution = 1
(*p* lags per *p* units of time)

# Static MLP for handling dynamics

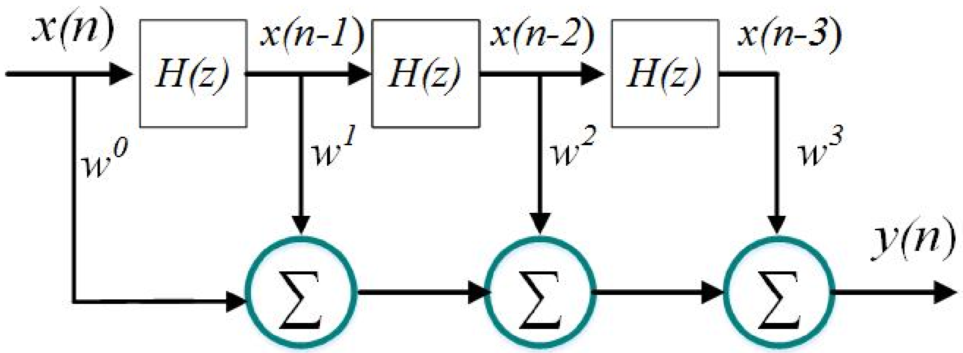
The use of a static MLP to account for temporal dimension

- short-term memory function
- nonlinear regression capabilities

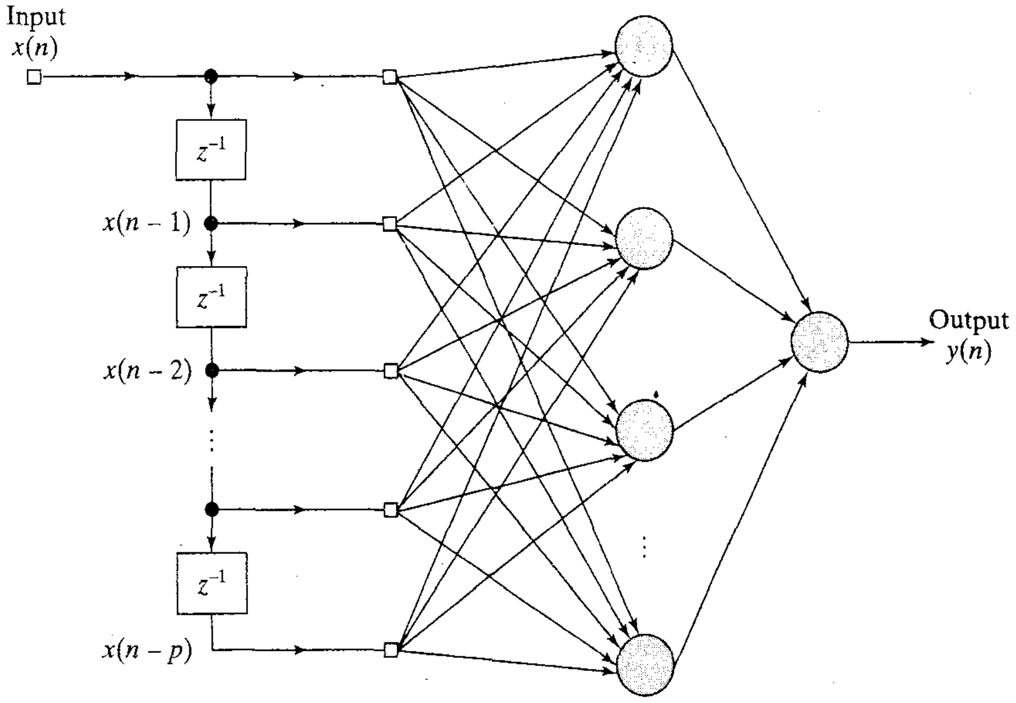
Tapped delay line memory



Generalized tapped delay line memory



# Learning approach to TLFN

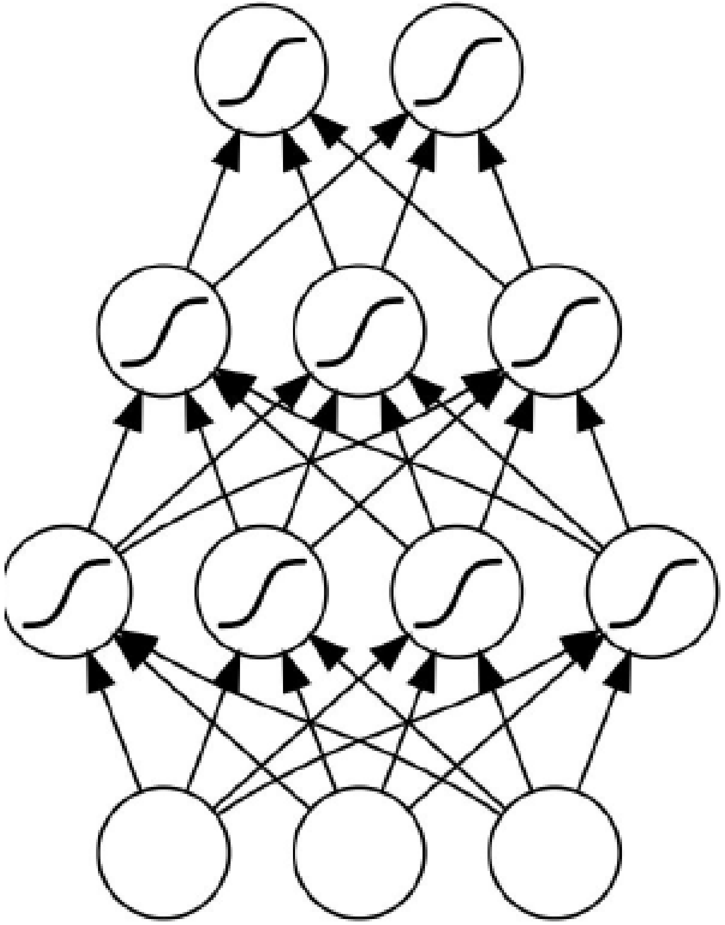


Backprop can be used with relatively simple *focused TLFNs* .

A general principle to unfold the network: form a large “static“ network, and apply backprop.

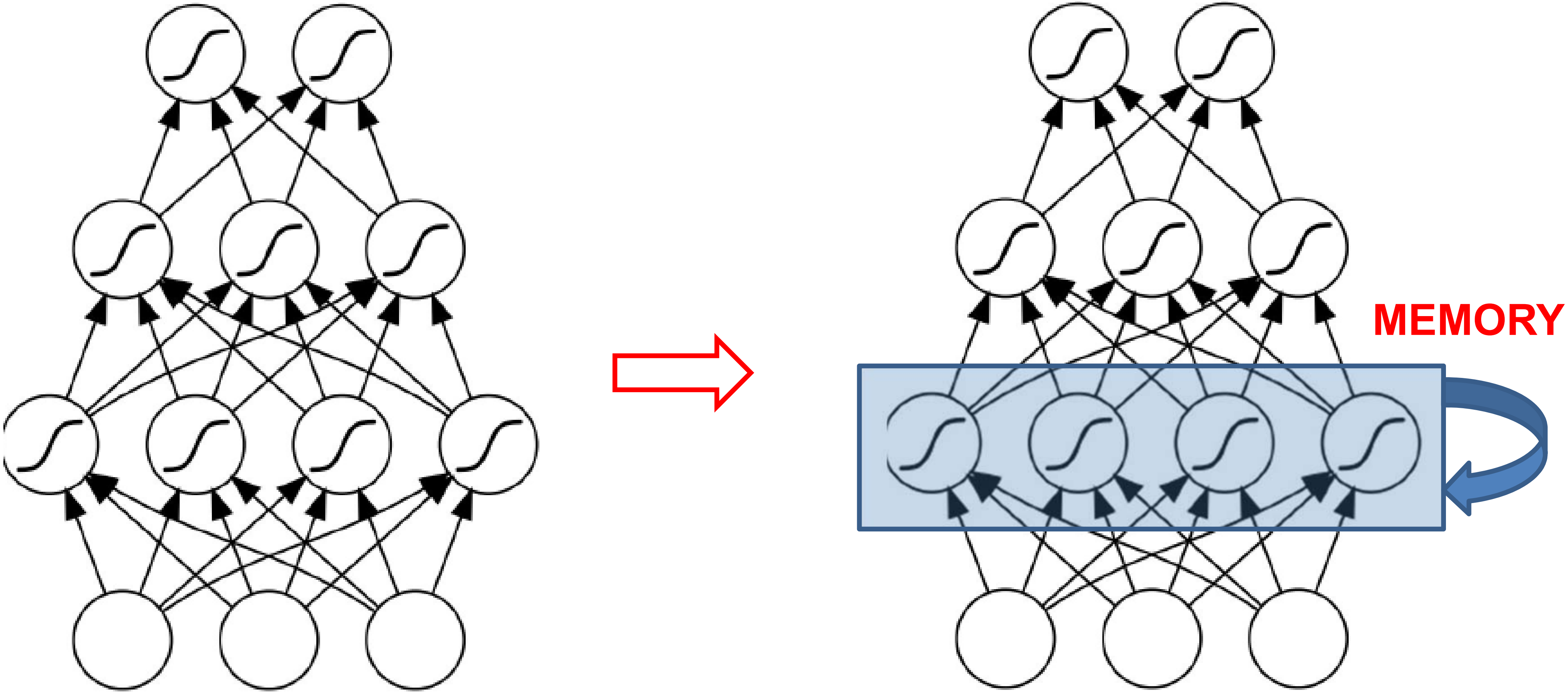
Haykin, 1999

# Recurrent neural networks (RNNs)



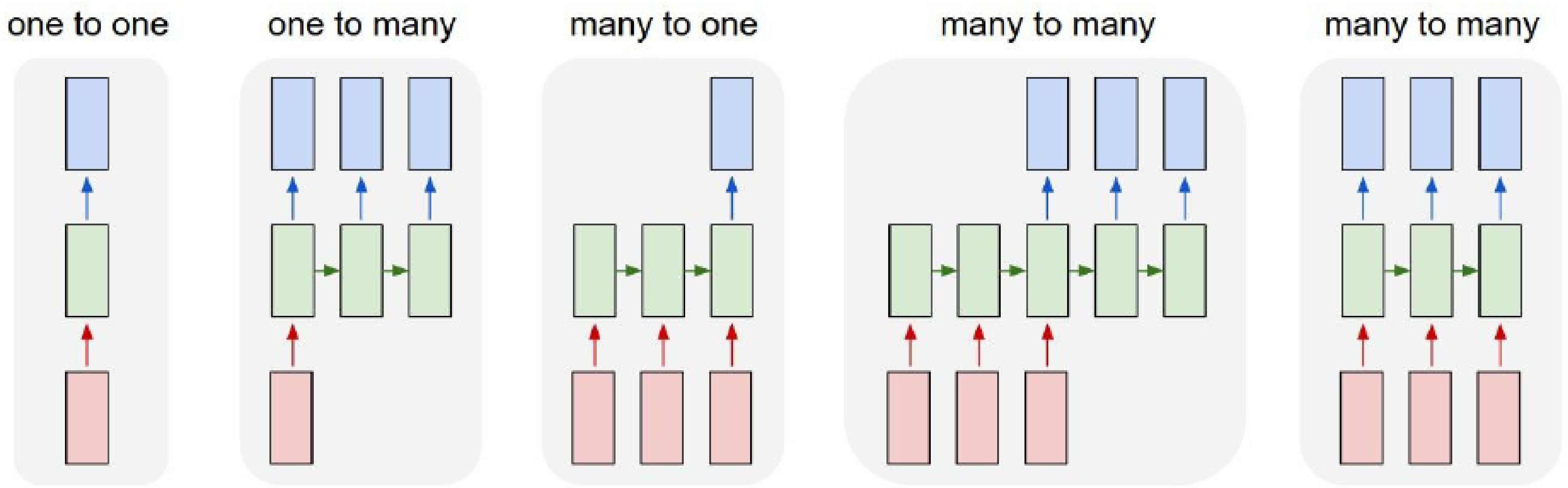
From MLP to ….

# Recurrent neural networks (RNNs)



From MLP to RNN

# Repertoire of recurrent architectures

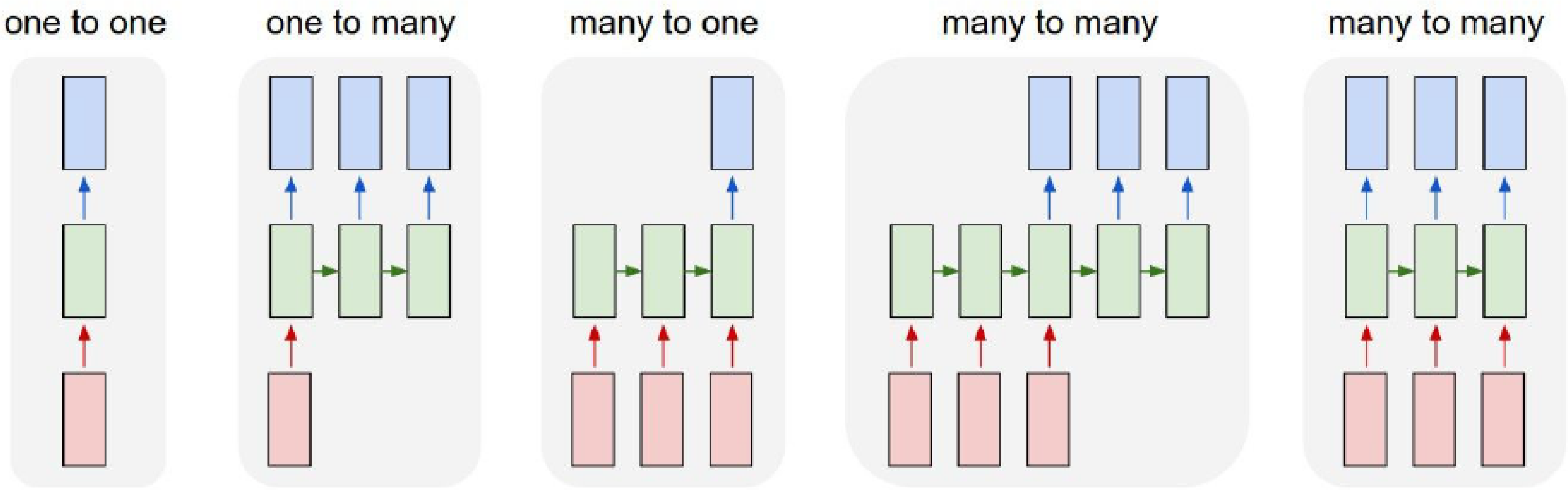


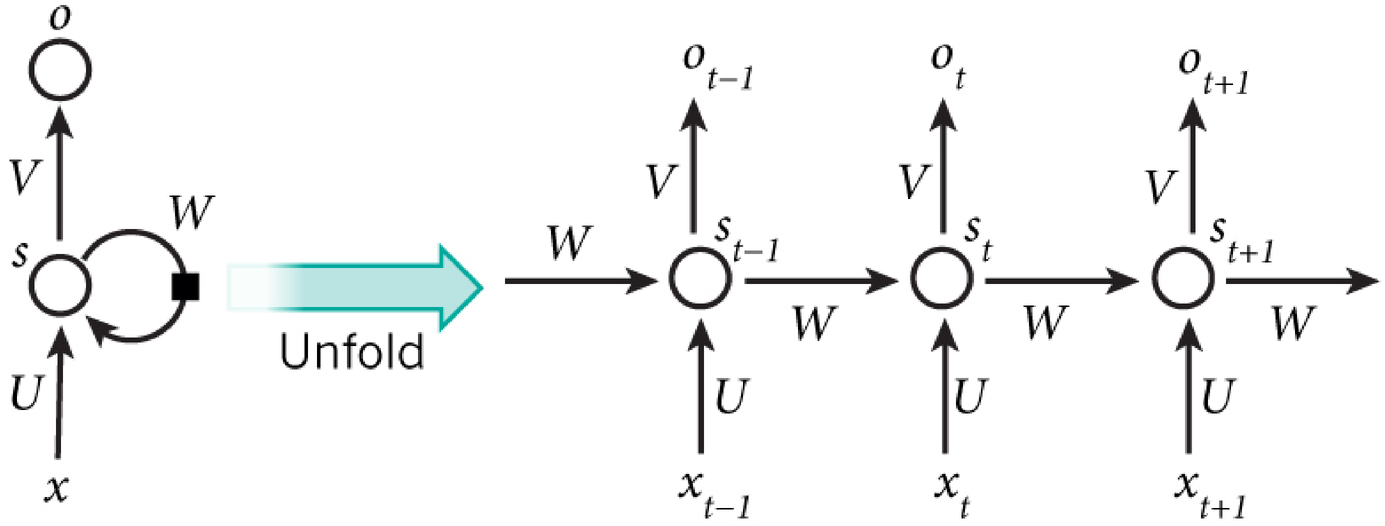
e.g. sequence generation

e.g. sentiment analysis

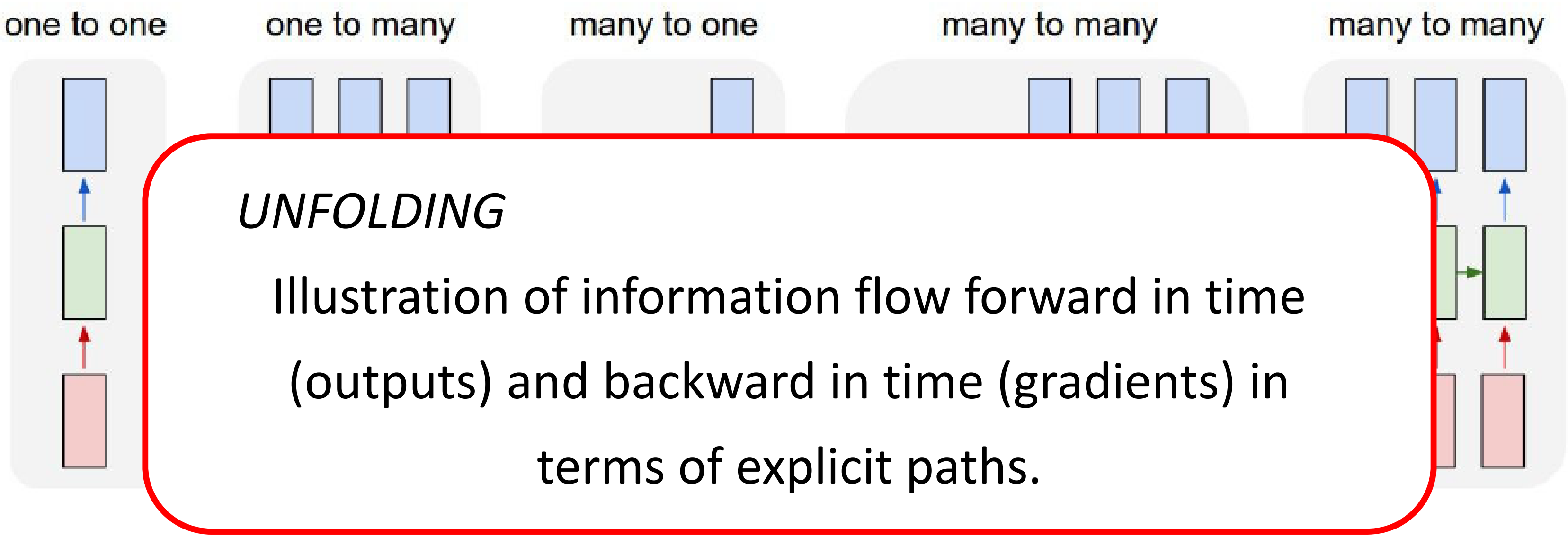
e.g. machine translation

# Repertoire of recurrent architectures, unfolding



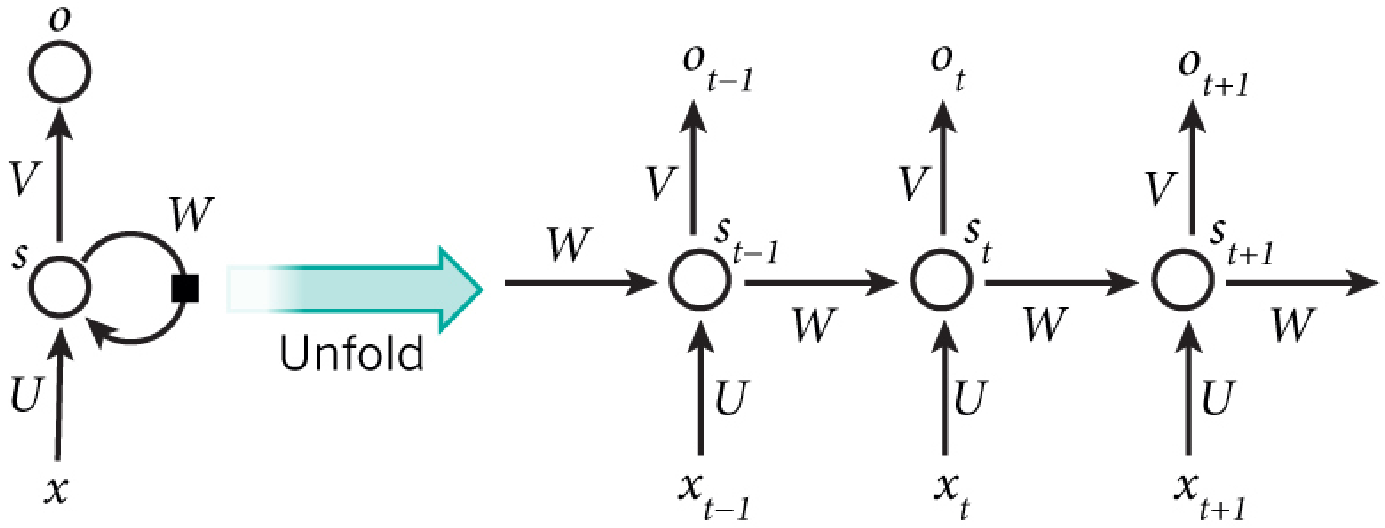


# Repertoire of recurrent architectures, unfolding

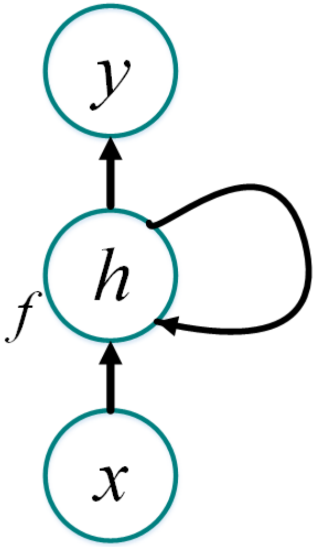


*UNFOLDING*

Illustration of information flow forward in time (outputs) and backward in time (gradients) in terms of explicit paths.



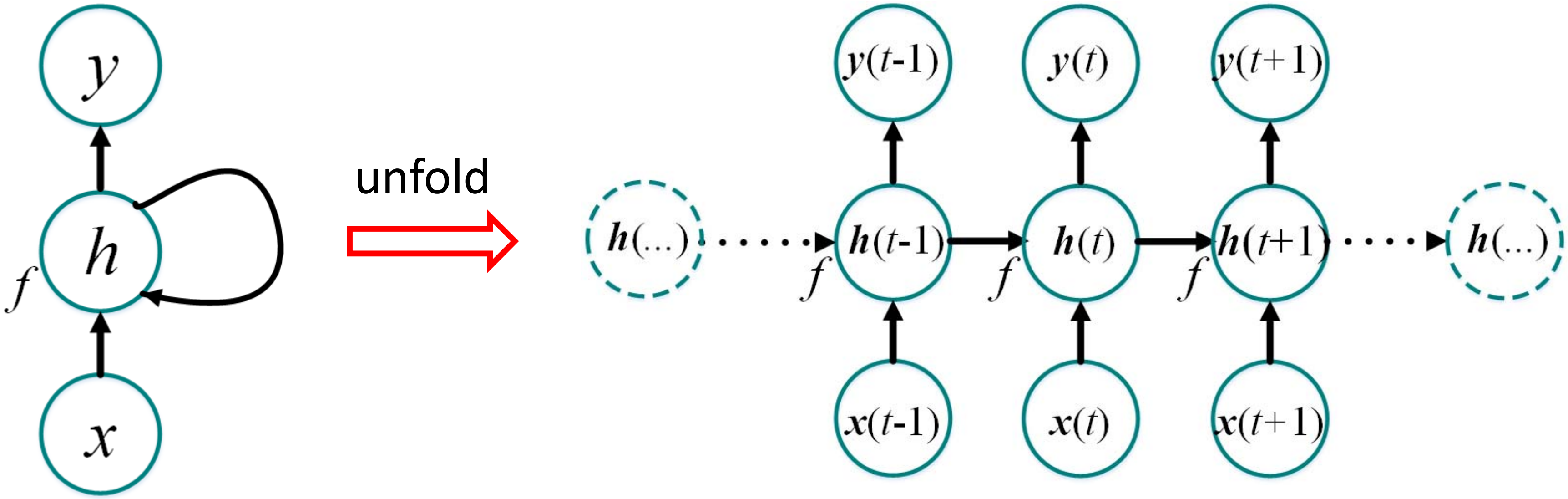
# Fundamental, vanilla RNN unit



$z^{-1}$ (single time unit delay, $h(t\text{-}1)$)

$$\boldsymbol{h}(t) = f(\boldsymbol{h}(t-1), \boldsymbol{x}(t), \boldsymbol{\theta})$$

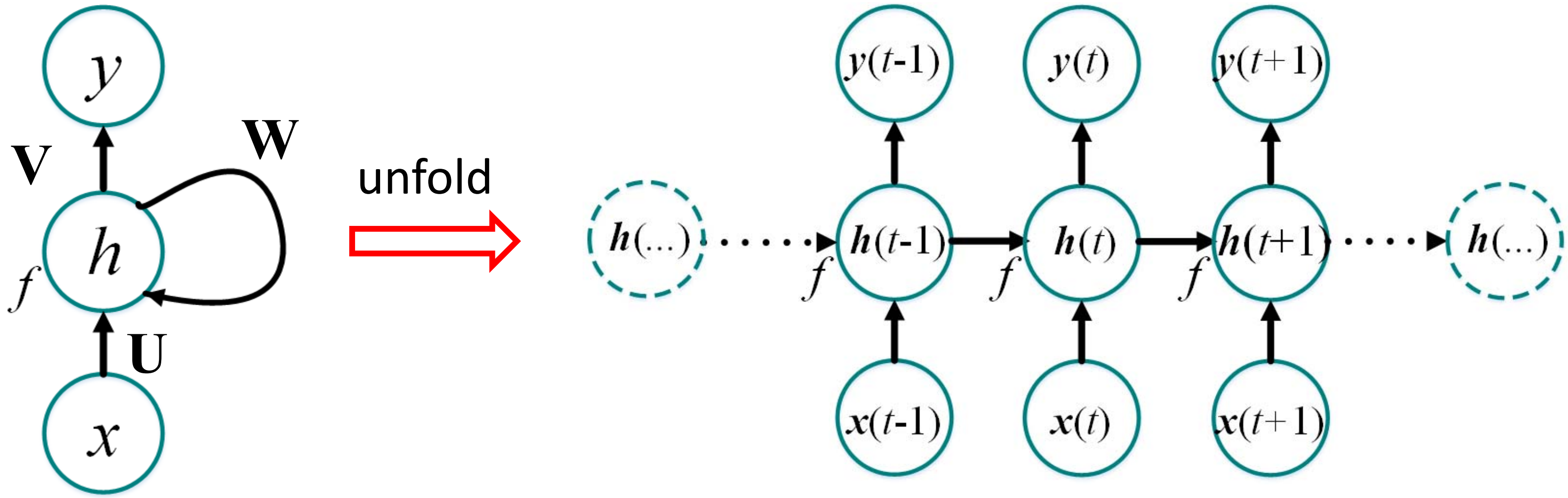
# Fundamental, vanilla RNN – unfolded



$$\boldsymbol{h}(t) = f(\boldsymbol{h}(t-1), \boldsymbol{x}(t), \boldsymbol{\theta})$$

In a canonical form it allows for modelling sequences of varying length (though there are problems of technical nature).

# Recurrent connections between hidden units

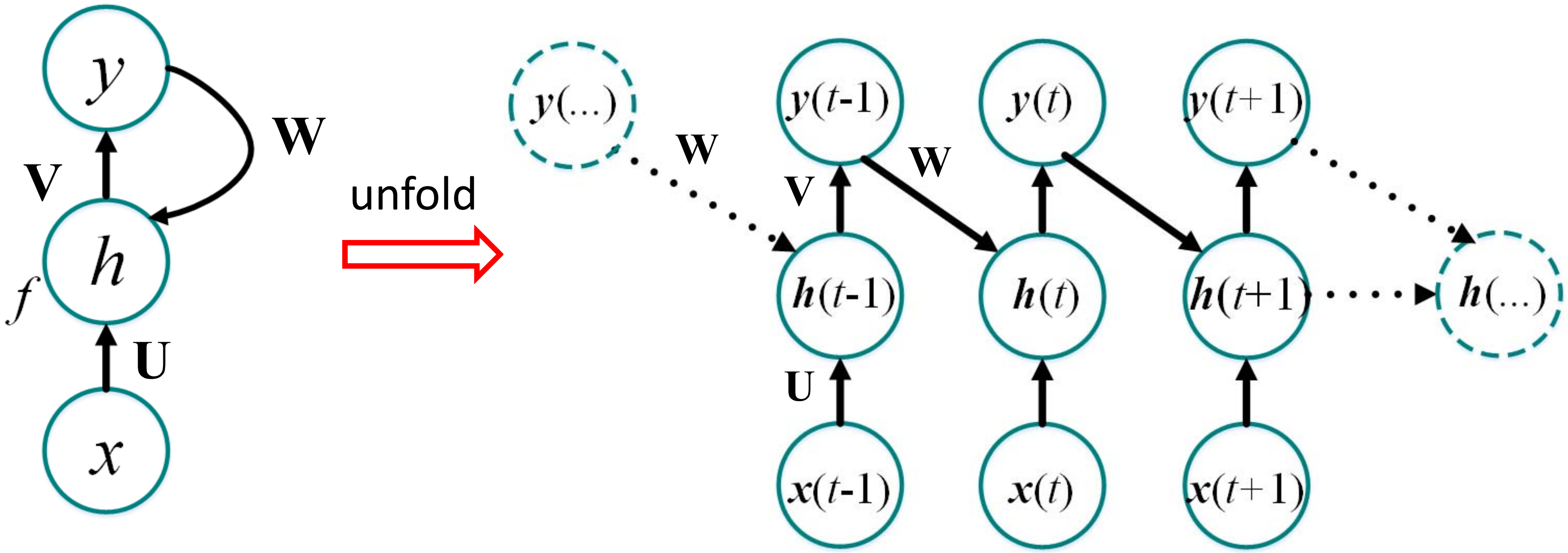


$$\mathbf{h}(t) = f(\mathbf{Wh}(t-1) + \mathbf{Ux}(t) + bias)$$

$$\mathbf{y}(t) = \mathbf{Vh}(t) + bias$$

state-space description

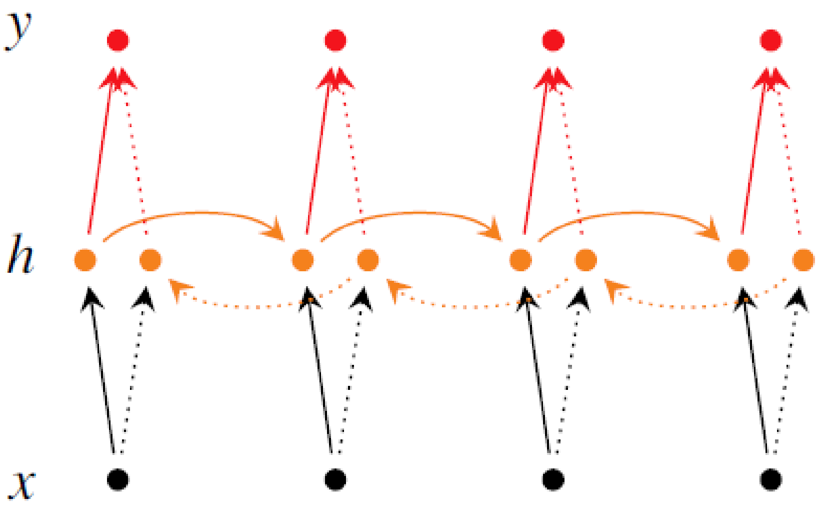
# Recurrent connection from output to hidden units



$$\mathbf{h}(t) = f(\mathbf{W}\mathbf{y}(t-1) + \mathbf{U}\mathbf{x}(t) + bias)$$

$$\mathbf{y}(t) = \mathbf{V}\mathbf{h}(t) + bias$$

# …. bidirectional neural networks



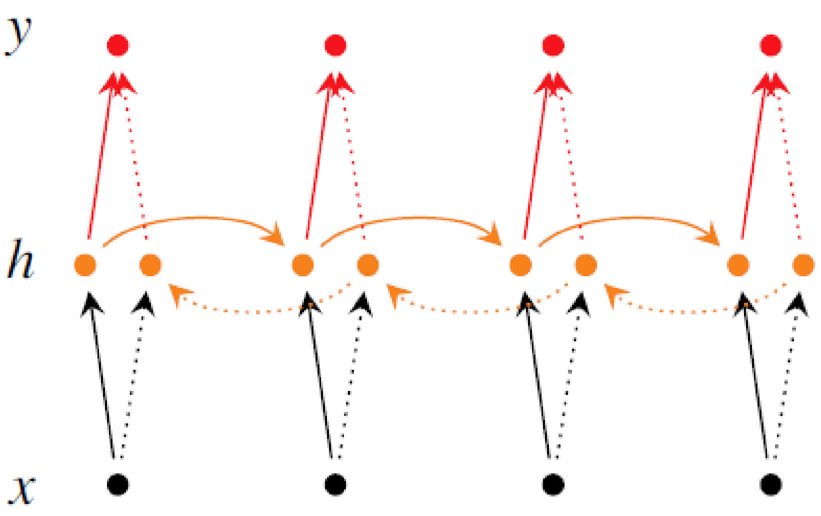
$$\overrightarrow{h}_t = f(\overrightarrow{W}x_t + \overrightarrow{V}\overrightarrow{h}_{t-1} + \overrightarrow{b})$$

$$\overleftarrow{h}_t = f(\overleftarrow{W}x_t + \overleftarrow{V}\overleftarrow{h}_{t+1} + \overleftarrow{b})$$

$$y_t = g(U[\overrightarrow{h}_t;\overleftarrow{h}_t] + c)$$

To incorporate information from words or phonemes both preceding and following, e.g. where there is clear dependency of phonemes/words on the following neighbouring phonemes/words

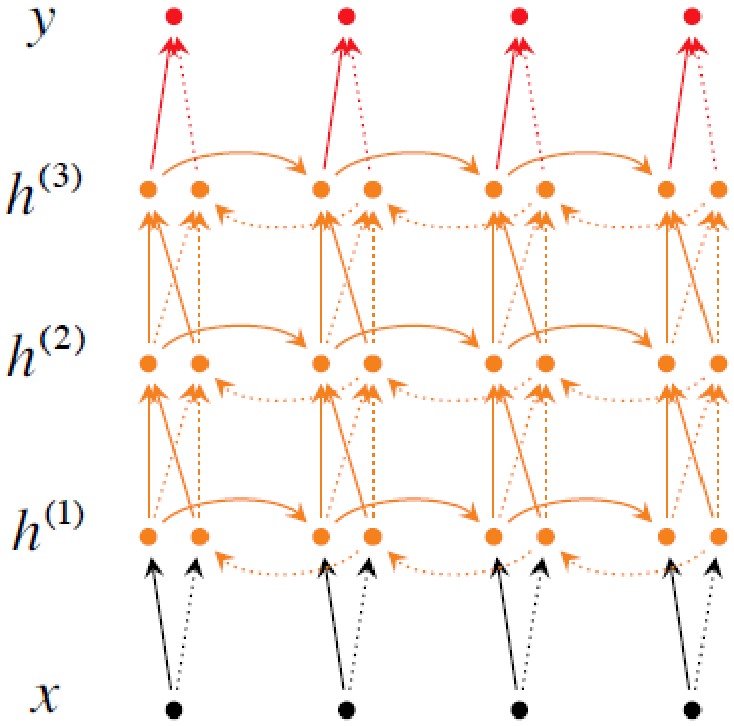
# Shallow vs deep bidirectional neural networks



$$\overrightarrow{h}_t = f(\overrightarrow{W} x_t + \overrightarrow{V}\overrightarrow{h}_{t-1} + \overrightarrow{b})$$

$$\overleftarrow{h}_t = f(\overleftarrow{W} x_t + \overleftarrow{V}\overleftarrow{h}_{t+1} + \overleftarrow{b})$$

$$y_t = g(U[\overrightarrow{h}_t;\overleftarrow{h}_t]+c)$$



$$\overrightarrow{h}_t^{(i)} = f(\overrightarrow{W}^{(i)} h_t^{(i-1)} + \overrightarrow{V}^{(i)}\overrightarrow{h}_{t-1}^{(i)} + \overrightarrow{b}^{(i)})$$

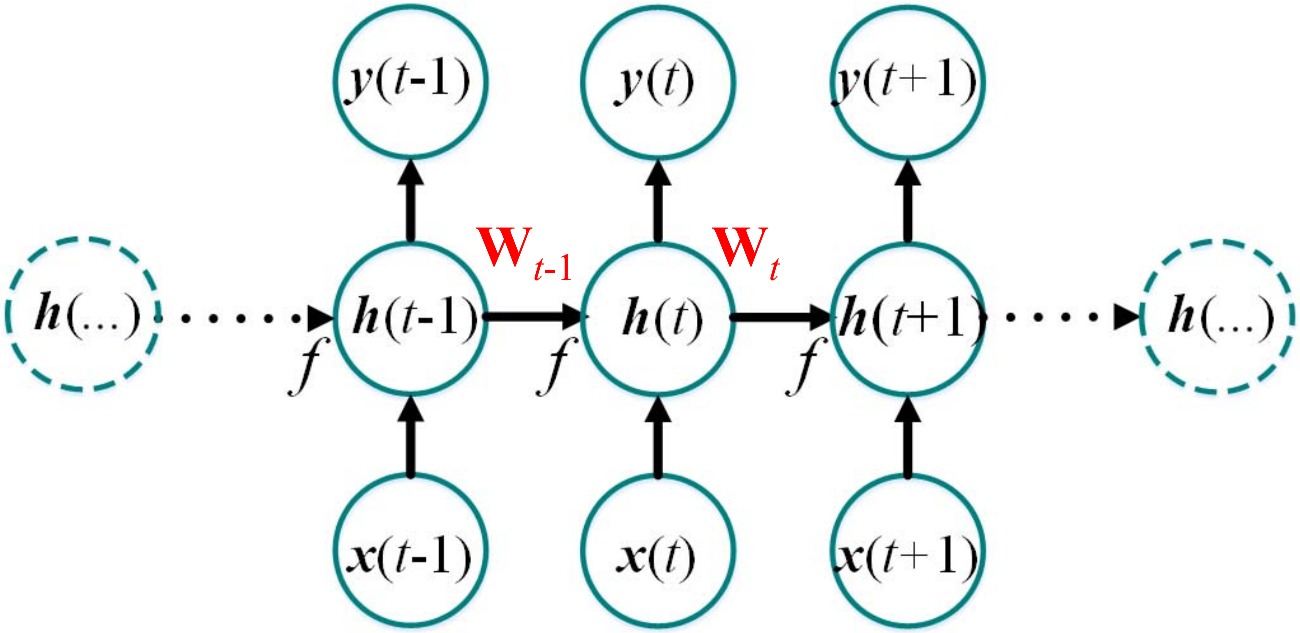
$$\overleftarrow{h}_t^{(i)} = f(\overleftarrow{W}^{(i)} h_t^{(i-1)} + \overleftarrow{V}^{(i)}\overleftarrow{h}_{t+1}^{(i)} + \overleftarrow{b}^{(i)})$$

$$y_t = g(U[\overrightarrow{h}_t^{(L)};\overleftarrow{h}_t^{(L)}]+c)$$

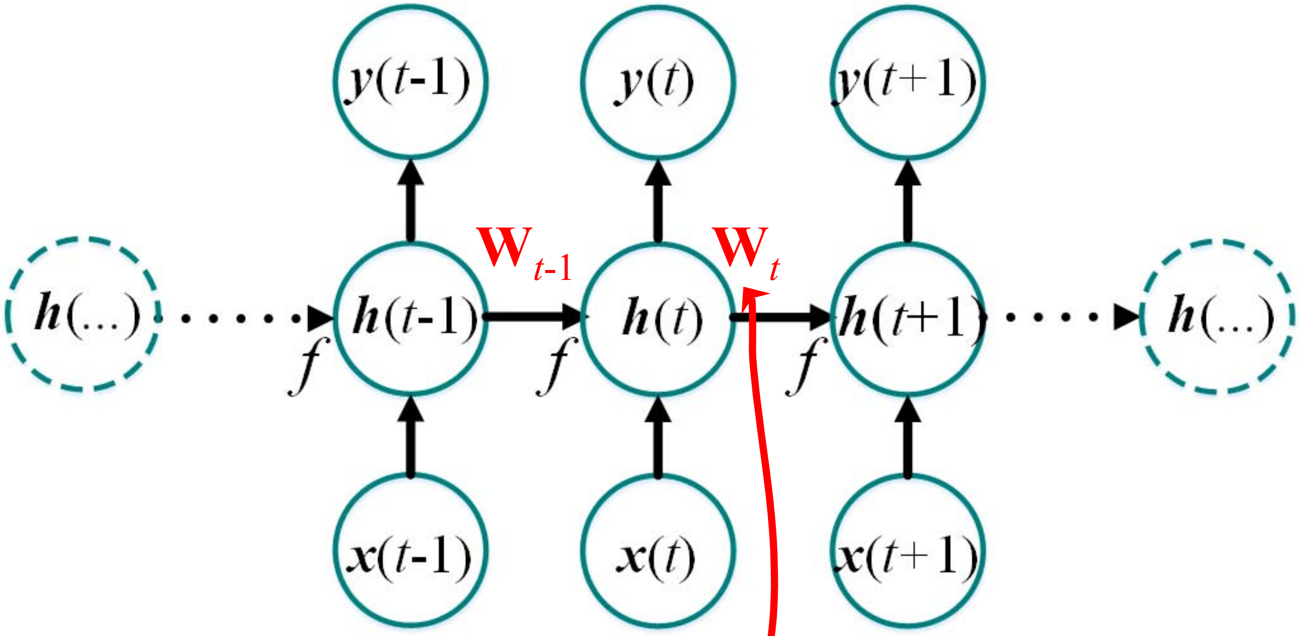
# Learning algorithms for RNNs

- Epochwise vs continuous training
  - "epoch" corresponds to a data sample – a sequence
  - RNN activity reset between epochs (in epochwise training)
- Backpropagation through time (unfolding into an MLP) can be both applied epochwise and in a continuous fashion

# Backpropagation through time



# Backpropagation through time

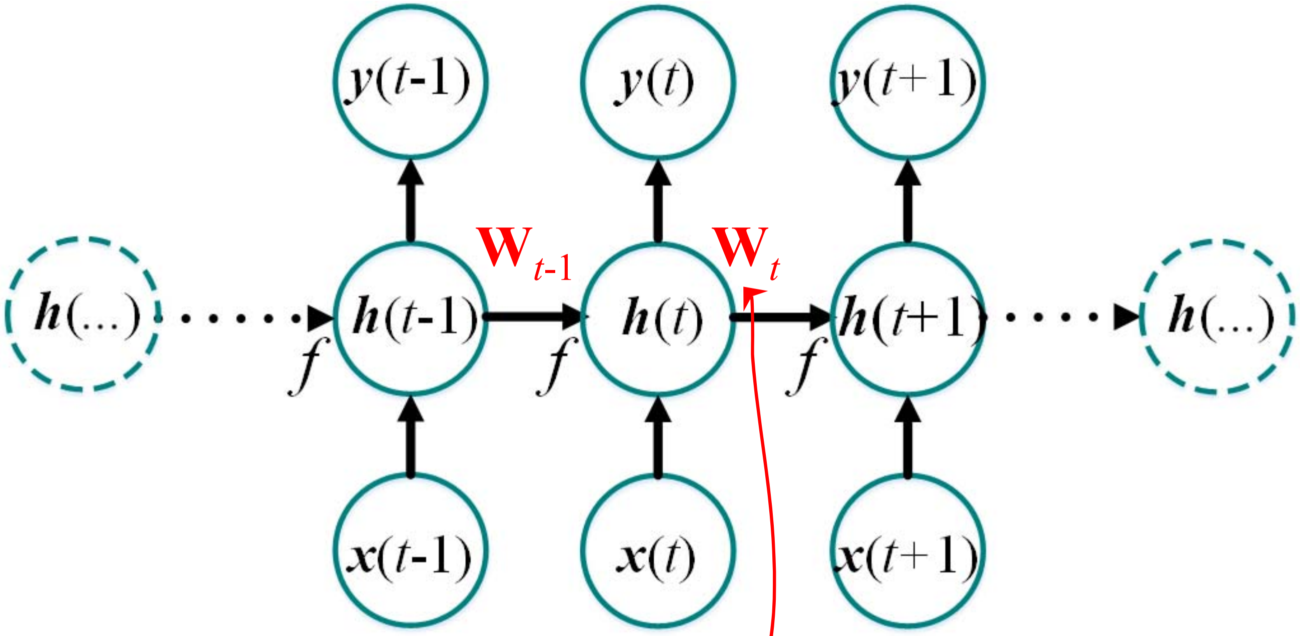


shared duplicated weights

Essentially,

$\mathbf{W}=\mathbf{W}_t$ , $\mathbf{W} = \mathbf{W}_{t\text{-}1}$ etc.

# Backpropagation through time



shared duplicated weights

Essentially,

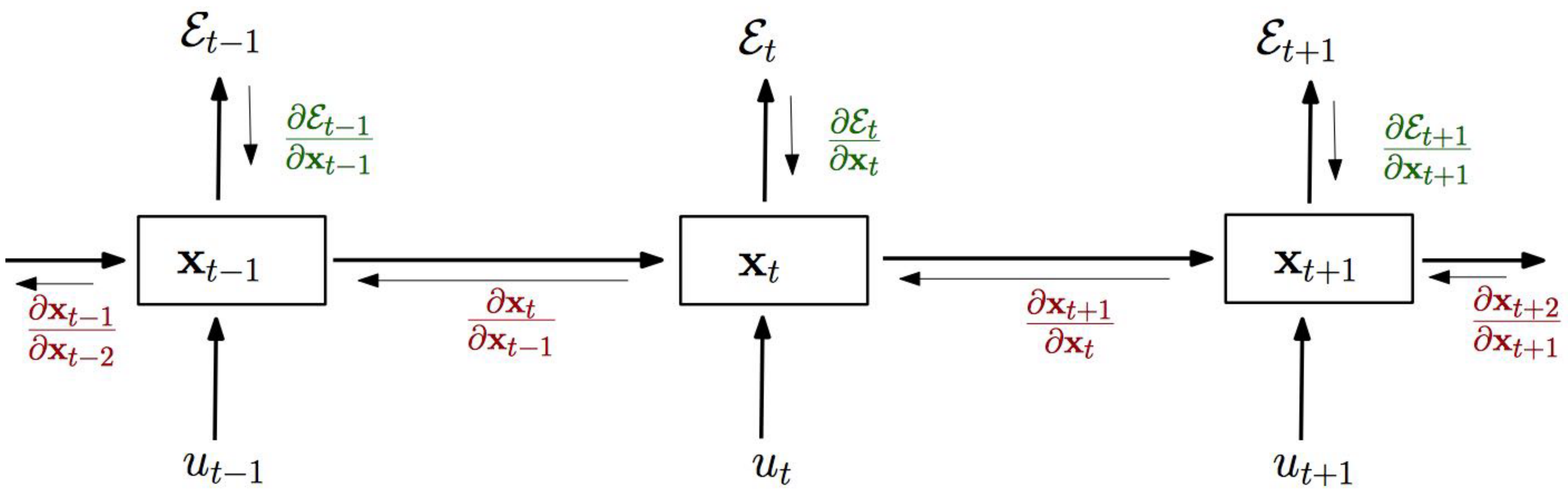
$\mathbf{W}{=}\mathbf{W}_t$ , $\mathbf{W} = \mathbf{W}_{t\text{-}1}$ etc.

So: $$\underbrace{\frac{\partial E}{\mathrm{W}} = \frac{\partial E}{\mathrm{W}_t} + \frac{\partial E}{\mathrm{W}_{t-1}} + \ldots.}_{\text{\#time steps in a training sample}}$$

# Backpropagation through time

For each pair of input-output sequences, the error is defined and $n$ refers to the index over the duration of sequences (not the number of samples):

$$E = \sum_{n=1}^{T} (d(n) - y(n))^2 = \sum_{n=1}^{T} \varepsilon_n^2$$


$\mathcal{E}_{t-1}$
$\mathcal{E}_t$
$\mathcal{E}_{t+1}$
$\frac{\partial \mathcal{E}_{t-1}}{\partial \mathbf{x}_{t-1}}$
$\frac{\partial \mathcal{E}_t}{\partial \mathbf{x}_t}$
$\frac{\partial \mathcal{E}_{t+1}}{\partial \mathbf{x}_{t+1}}$
$\mathbf{x}_{t-1}$
$\mathbf{x}_t$
$\mathbf{x}_{t+1}$
$\frac{\partial \mathbf{x}_{t-1}}{\partial \mathbf{x}_{t-2}}$
$\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_{t-1}}$
$\frac{\partial \mathbf{x}_{t+1}}{\partial \mathbf{x}_t}$
$\frac{\partial \mathbf{x}_{t+2}}{\partial \mathbf{x}_{t+1}}$
$u_{t-1}$
$u_t$
$u_{t+1}$

Please note: According to our earlier notation, $\boldsymbol{x}$ should be $\boldsymbol{h}$ and $u$ should be $x$.

# Backpropagation through time

For each pair of input-output sequences, the error is defined and $n$ refers to the index over the duration of sequences (not the number of samples):

$$E = \sum_{n=1}^{T} (d(n) - y(n))^2 = \sum_{n=1}^{T} \varepsilon_n^2$$

We can also think of this training algorithm in the time domain (*Hinton, 2013*):

- FORWARD PASS: a stack of the activities of all the units at each time step.

- BACKWARD PASS: activities are peeled off the stack to compute the error derivatives at each time step.

- THEN we add together the derivatives at all the different times for each weight.

# Backpropagation through time

For each pair of input-output sequences, the error is defined and $n$ refers to the index over the duration of sequences (not the number of samples):

$$E = \sum_{n=1}^{T} (d(n) - y(n))^2 = \sum_{n=1}^{T} \varepsilon_n^2$$

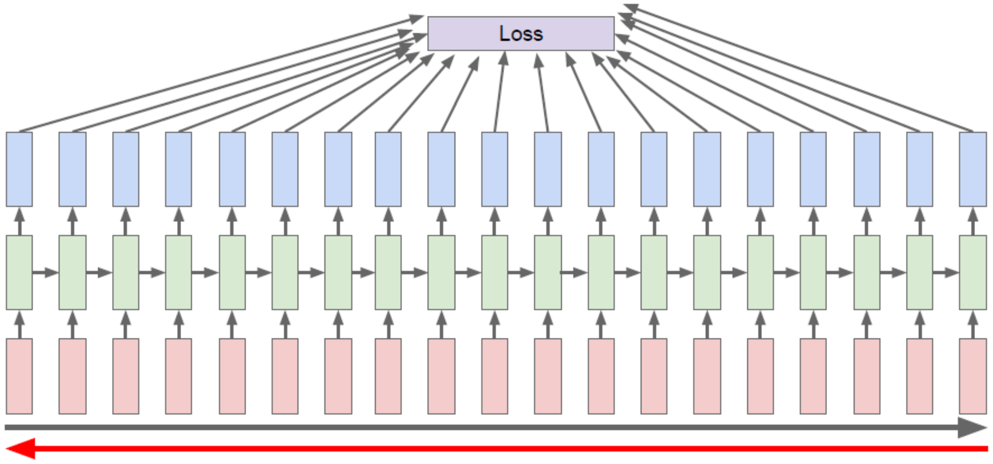
Comments:

- epoch refers here to a single pair of input-output sequences
- if there are backprojections from the output to hidden layer, teacher output *d*(*n*) can be used in the computation of activations in layer *n*+1 in the forward pass
    - teacher forcing is likely to speed the convergence
    - if it is exploited for the trained network however it may exhibit instability
- BPTT does not scale too well and may not converge even to the local minimum
- BPTT may result in instability

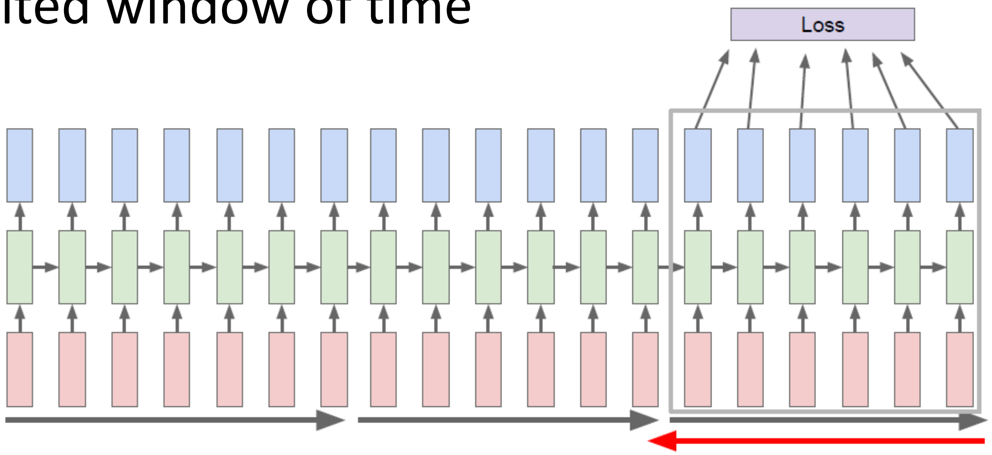
# Backpropagation through time – online with truncation

## Truncated BPTT for “online” learning

- if there are no batches and the network is supposed to work continuously, the number of recursive steps for weight updates has to be finite
- beyond the truncation there is no memory effect



with *truncation* backpropagate loss within a limited window of time



*For more details, please see Haykin and Goodfellow et al.*

# Other learning algorithms for RNNs

- Real-time recurrent learning (Williams & Zipser, 1989)
  - the exact gradients are calculated and the synaptic updates are made at every step during network's processing stage
  - very high computational cost
- Kalman filters (*optimal filtering*)
  - solid theory formulated in the state-space concepts
  - rather than instantaneously estimate gradients, the network state is recursively estimated based on input data
  - the network has to be linearised
- Extended Kalman filter (*decoupled*)

# Long-term dependencies problem

Vanishing (more rarely exploding) gradients for recurrent networks

$$\boldsymbol{h}(t) = \mathbf{W}^{\mathrm{T}}\boldsymbol{h}(t-1) \Rightarrow \boldsymbol{h}(t) = (\mathbf{W}^{t})^{\mathrm{T}}\boldsymbol{h}(0)$$

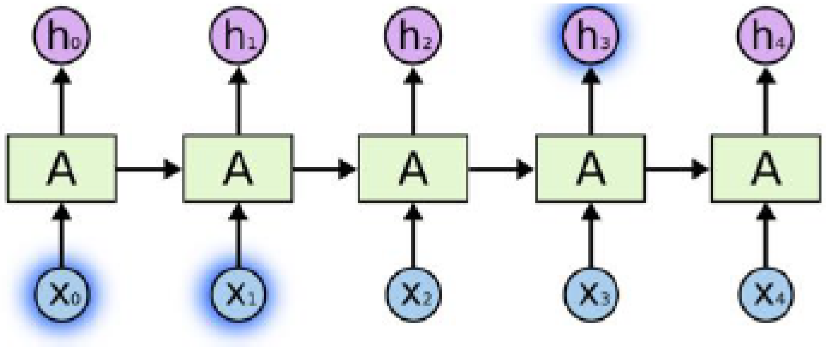
$$\mathbf{W} = \mathbf{Q\Lambda Q}^{\mathrm{T}} \Rightarrow \boldsymbol{h}(t) = \mathbf{Q}^{\mathrm{T}}\mathbf{\Lambda}^{t}\mathbf{Q}\,\boldsymbol{h}(0)$$

Eigenvalues are usually less than 1

# Long-term dependencies problem

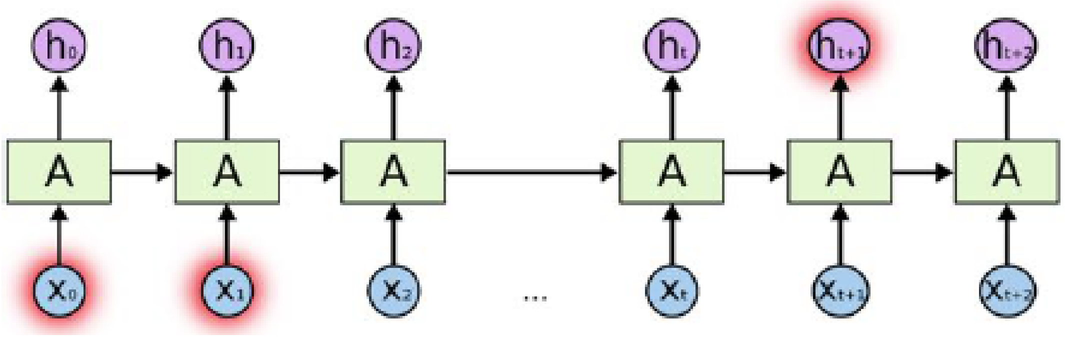
Handling (predicting) close words:

"The *clouds* are in the *sky*."



Problem with the need for wider context:

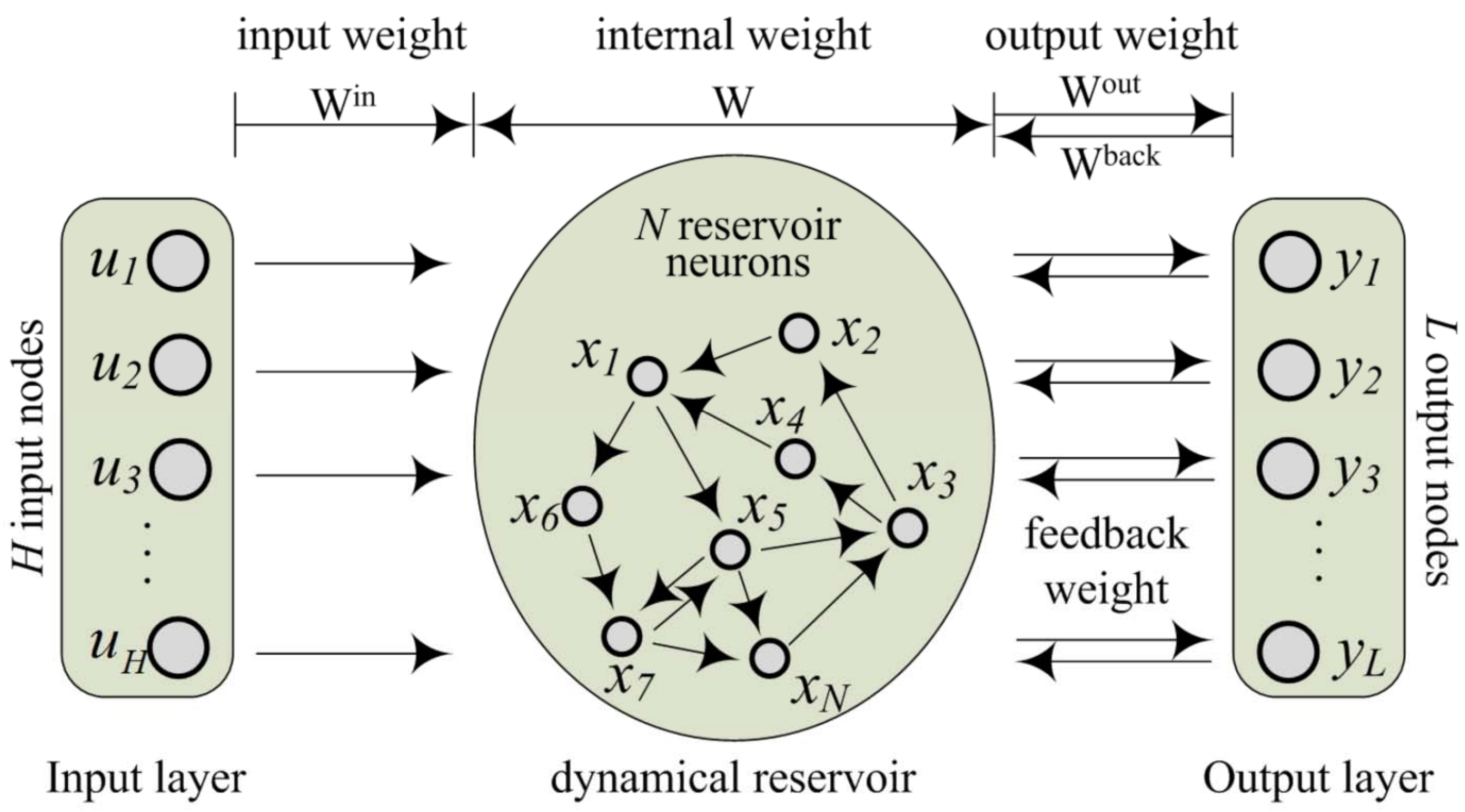
"I grew up in *France*… I speak fluent *French*."



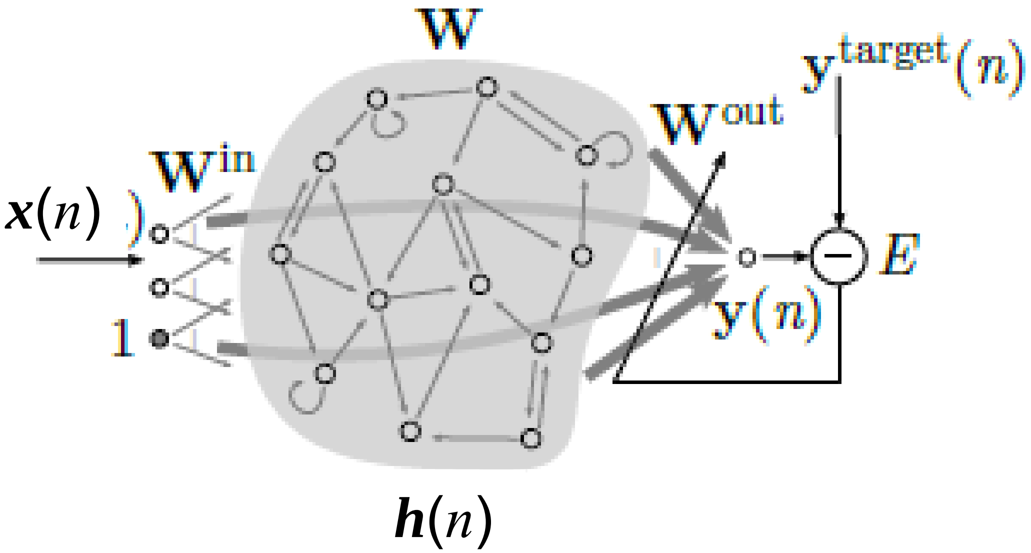
# Reservoir computing

Echo state network

(non-spiking version of liquid state machine)



# Reservoir computing – fundamentals



biases

$$\boldsymbol{y}(n) = \mathbf{W}^{out}\left[1;\, \boldsymbol{h}(n);\, \boldsymbol{x}(n)\right]$$

$$\tilde{\boldsymbol{h}}(n) = \tanh(\mathbf{W}_{in}\left[1; \boldsymbol{x}(n)\right] + \mathbf{W}\boldsymbol{h}(n-1))$$

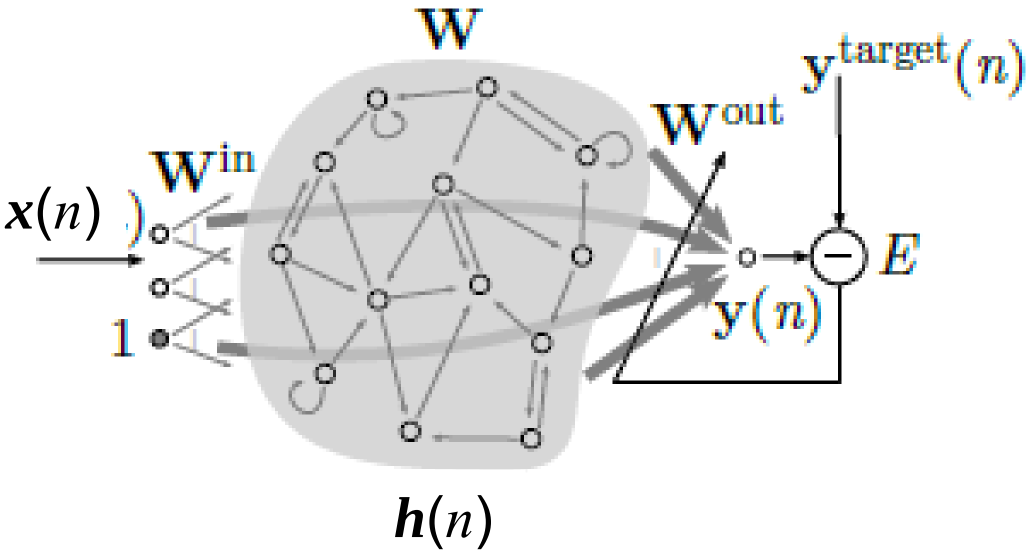
$$\boldsymbol{h}(n) = (1-\alpha)\boldsymbol{h}(n-1) + \alpha\tilde{\boldsymbol{h}}(n)$$

$$\mathbf{W} \in \mathbb{R}^{N_x \times N_x}$$

$$\mathbf{W}_{in} \in \mathbb{R}^{N_x \times (1+N_u)}$$

$$\mathbf{W}^{out} \in \mathbb{R}^{N_y \times (1+N_h+N_x)}$$

# Reservoir computing – fundamentals



biases

$$\boldsymbol{y}(n) = \mathbf{W}^{out}\left[1;\, \boldsymbol{h}(n);\, \boldsymbol{x}(n)\right]$$

$$\tilde{\boldsymbol{h}}(n) = \tanh(\mathbf{W}_{in}\left[1; \boldsymbol{x}(n)\right] + \mathbf{W}\boldsymbol{h}(n-1) + \mathbf{W}^{fb}\boldsymbol{y}(n-1))$$

$$\boldsymbol{h}(n) = (1-\alpha)\boldsymbol{h}(n-1) + \alpha\tilde{\boldsymbol{h}}(n)$$

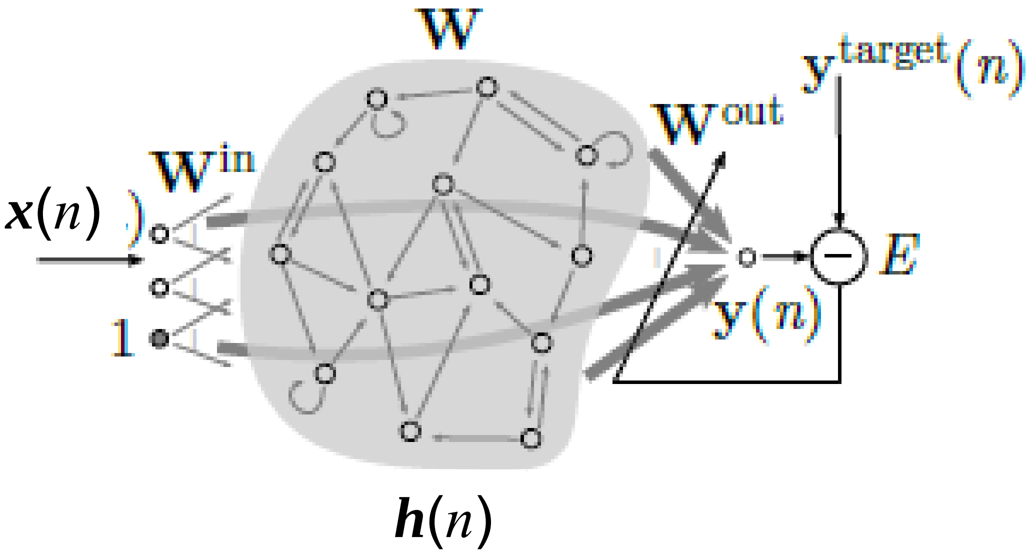
$$\mathbf{W} \in \mathbb{R}^{N_x \times N_x}$$

$$\mathbf{W}_{in} \in \mathbb{R}^{N_x \times (1+N_u)}$$

$$\mathbf{W}^{out} \in \mathbb{R}^{N_y \times (1+N_h+N_x)}$$

extra *feedback* projections from the output to the hidden layer

# Reservoir computing – fundamentals



biases

$$\boldsymbol{y}(n) = \mathbf{W}^{out}\left[1;\, \boldsymbol{h}(n);\, \boldsymbol{x}(n)\right]$$

$$\tilde{\boldsymbol{h}}(n) = \tanh(\mathbf{W}_{in}\left[1; \boldsymbol{x}(n)\right] + \mathbf{W}\boldsymbol{h}(n-1) + \mathbf{W}^{fb}\boldsymbol{y}(n-1))$$

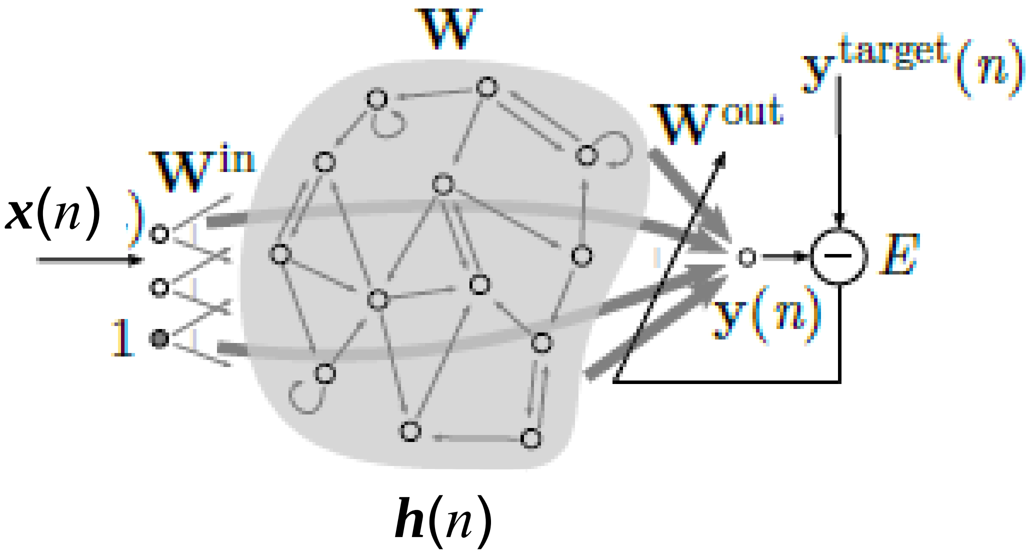
$$\boldsymbol{h}(n) = (1-\alpha)\boldsymbol{h}(n-1) + \alpha\tilde{\boldsymbol{h}}(n)$$

Could be used for providing feedback for training – *teacher forcing*:

$$y(n) = y^{\text{target}}(n)$$

extra *feedback* projections from the output to the hidden layer

# Reservoir computing – fundamentals



biases

$$\boldsymbol{y}(n) = \mathbf{W}^{out}\left[1;\ \boldsymbol{h}(n);\ \boldsymbol{x}(n)\right]$$

$$\tilde{\boldsymbol{h}}(n) = \tanh(\mathbf{W}_{in}\left[1;\boldsymbol{x}(n)\right] + \mathbf{W}\boldsymbol{h}(n-1) + \mathbf{W}^{fb}\boldsymbol{y}(n-1))$$

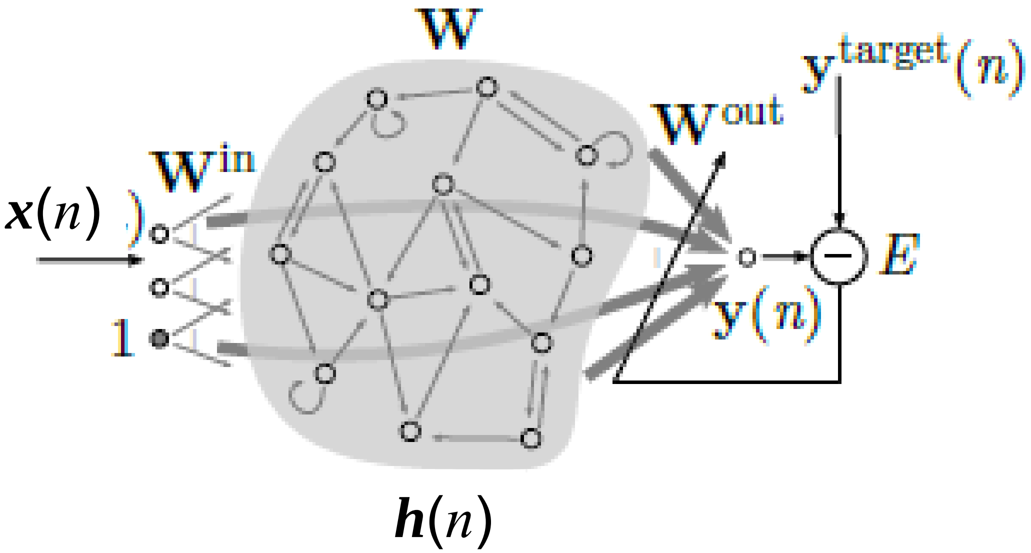
$$\boldsymbol{h}(n) = (1-\alpha)\boldsymbol{h}(n-1) + \alpha\tilde{\boldsymbol{h}}(n)$$

HOWEVER, in on-line learning the use of $y(n)$ for feedback is preferred for the stability.

extra *feedback* projections from the output to the hidden layer

# Reservoir properties

**Reservoir serves as a memory (temporal context) and a nonlinear expansion**



Key parameters: .

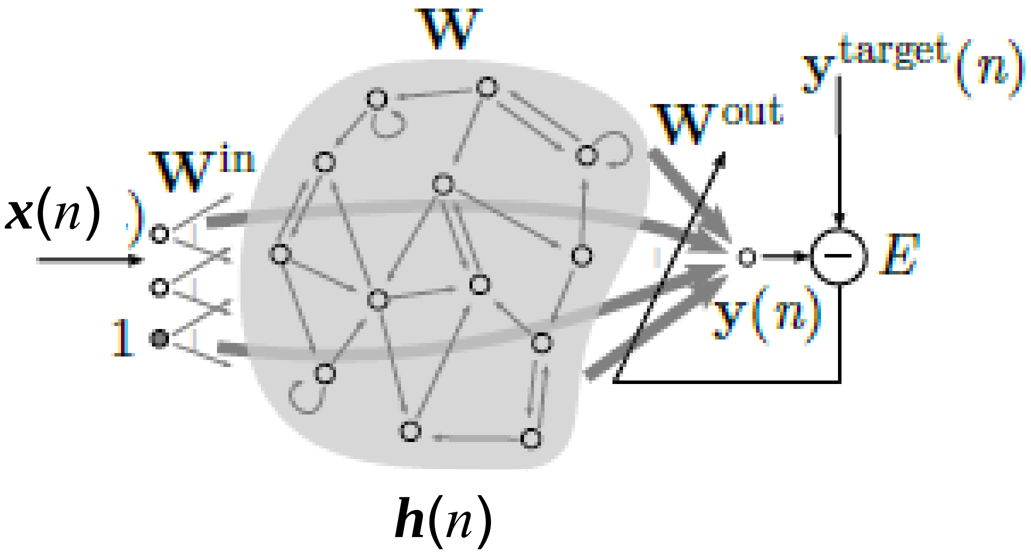
- size ($N_x$) (the bigger the better, even in the order of 10k)

- sparsity (2-20%) and the distribution of nonzero elements (uniform distribution)

- spectral radius, $\rho$ (less than 1 to be near the edge of stability)

# Reservoir properties

**Reservoir serves as a memory (temporal context) and a nonlinear expansion**



Key parameters: .

- size ($N_x$) (the bigger the better, even in the order of 10k)

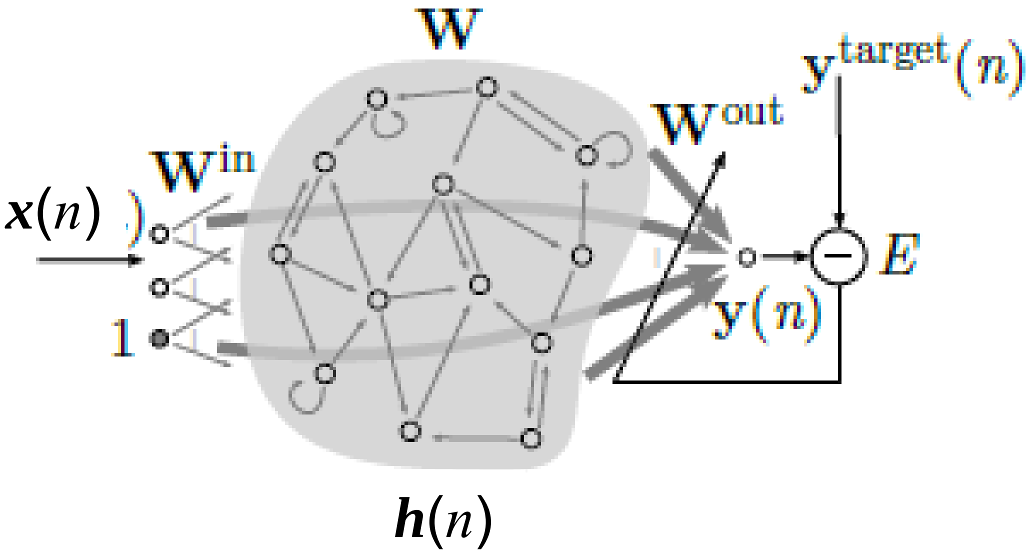
- sparsity (2-20%) and the distribution of nonzero elements (uniform distribution)

- spectral radius, $\rho$ (less than 1 to be near the edge of stability)

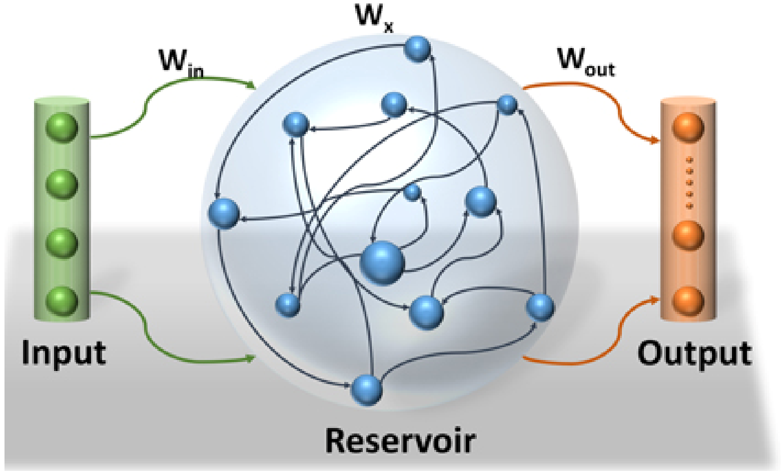
For nonlinear networks, the system can still be contractive and stable for $\rho$>1.
So, commonly in practice, $\rho \approx 3$.

# Reservoir properties

**Reservoir serves as a memory (temporal context) and a nonlinear expansion**



Sparse, random and fixed connections in the reservoir…



Key parameters: .

- size ($N_x$)
- sparsity (2-20%)
- spectral radius, $\rho$

….with "leaky" units

Leak modulated by $\alpha$

$$h(t) = \alpha h(t-1) + (1-\alpha)x(t)$$

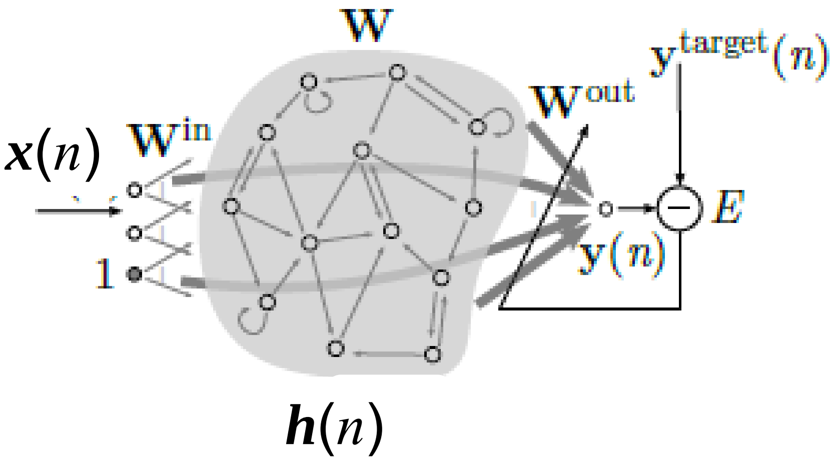
# Training readouts

- One-shot learning with least mean square (LMS) error approaches
  - the design matrix is usually overdetermined -> ridge regression (regularization):

$$\mathbf{Y}^{target} = \mathbf{W}^{out}\mathbf{H} \quad \rightarrow \quad \underline{\mathbf{W}^{out} = \mathbf{Y}^{target}\mathbf{H}^{\mathrm{T}}\left(\mathbf{H}\mathbf{H}^{\mathrm{T}} + \beta\mathbf{I}\right)}^{-1}$$

  - be careful with extremely large values in **W** as they may indicate problems with stability
- Online learning with, for example, *recursive* LMS
- The need to deal with initial transients, esp. for long sequences
- Possible use of teacher forcing (forcing output data through backprojections)

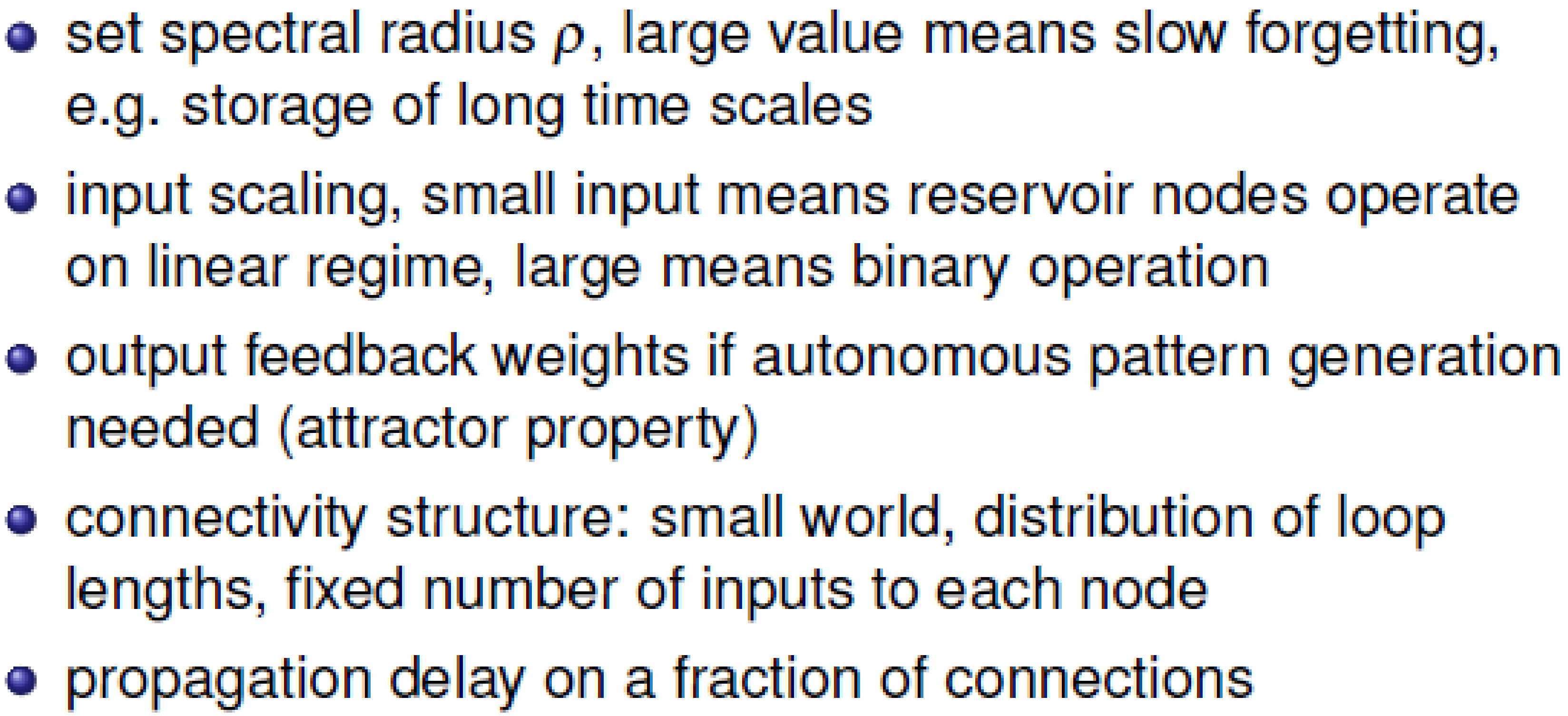
# Reservoir computing – overall recipe



## Recipe

1. Generation of the dynamic reservoir ($\boldsymbol{W}_{in}$, $\mathbf{W}$)
2. Application of inputs, $\boldsymbol{x}(n)$, and collecting the corresponding activation states, $\boldsymbol{h}(n)$.
3. Computation of the linear output weights from the reservoir with a linear regression approach (MSE error to be minimised).
4. Use of the RNN for new data – predictions.

# Reservoir computing



- set spectral radius $\rho$, large value means slow forgetting, e.g. storage of long time scales
- input scaling, small input means reservoir nodes operate on linear regime, large means binary operation
- output feedback weights if autonomous pattern generation needed (attractor property)
- connectivity structure: small world, distribution of loop lengths, fixed number of inputs to each node
- propagation delay on a fraction of connections

# Long short-term memory (LSTM)

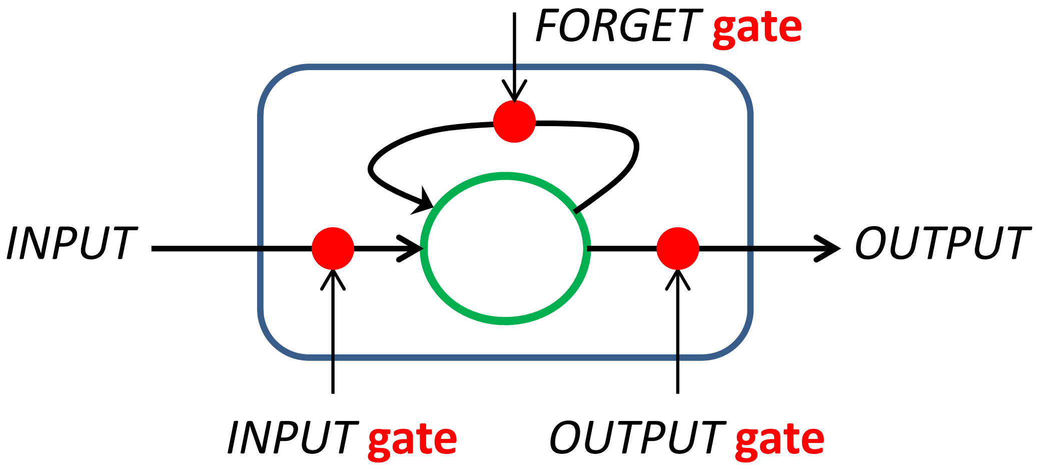
## Main motivation behind LSTMs

- vanishing gradients when using backprop through time for RNNs
- poor capacity to handle long-term dependencies

# Long short-term memory (LSTM)

## Main motivation behind LSTMs

- vanishing gradients when using backprop through time for RNNs
- poor capacity to handle long-term dependencies

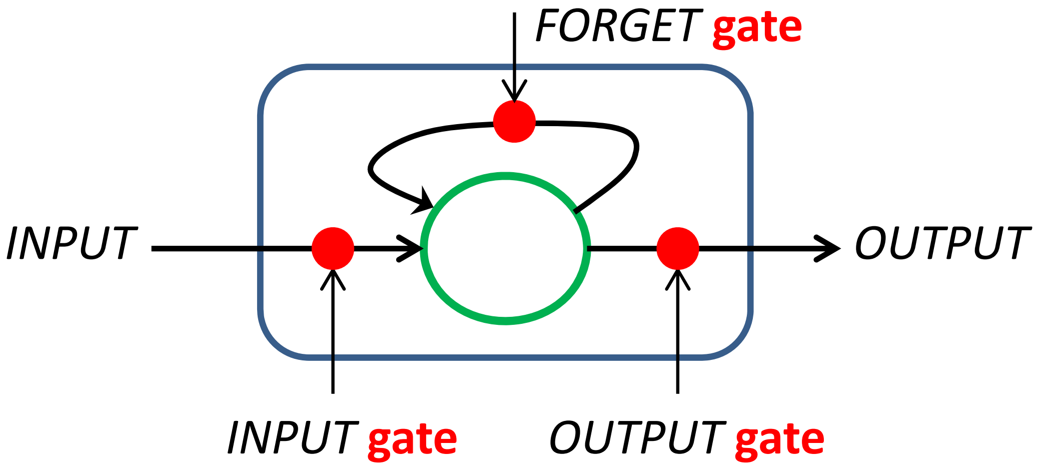


The key idea is to have a “memory cell” capable of keeping the state over time

# Long short-term memory (LSTM)

## Main motivation behind LSTMs

- vanishing gradients when using backprop through time for RNNs
- poor capacity to handle long-term dependencies



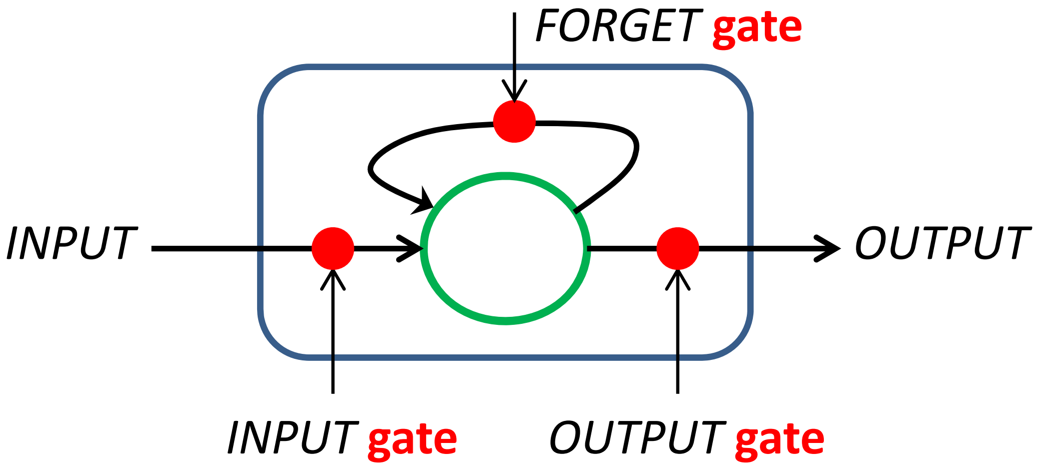
The key idea is to have a “memory cell” capable of keeping the state over time

forget gate acts like a “leak” with tuneable gain $h(t) = \alpha h(t-1) + (1-\alpha) x(t)$

# Long short-term memory (LSTM)

## Main motivation behind LSTMs

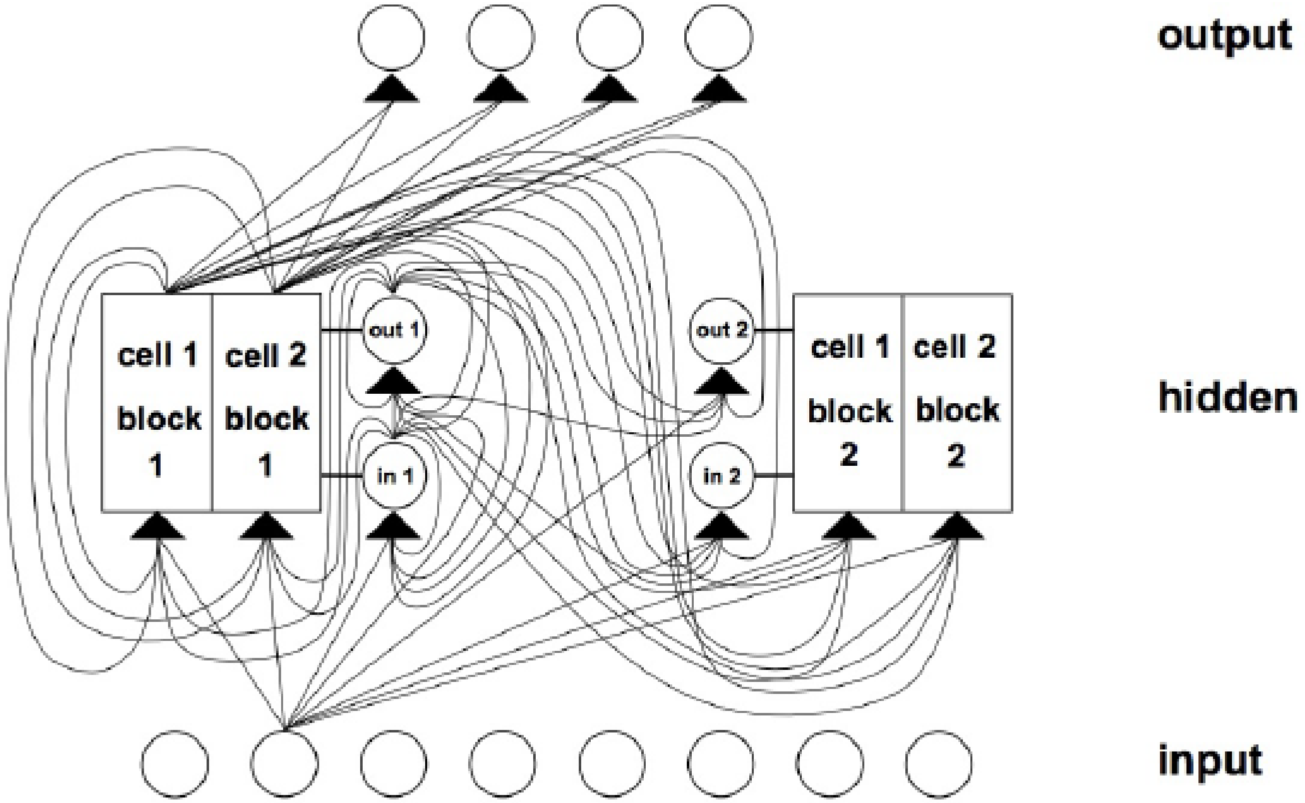
- vanishing gradients when using backprop through time for RNNs
- poor capacity to handle long-term dependencies



The key idea is to have a "memory cell" capable of keeping the state over time

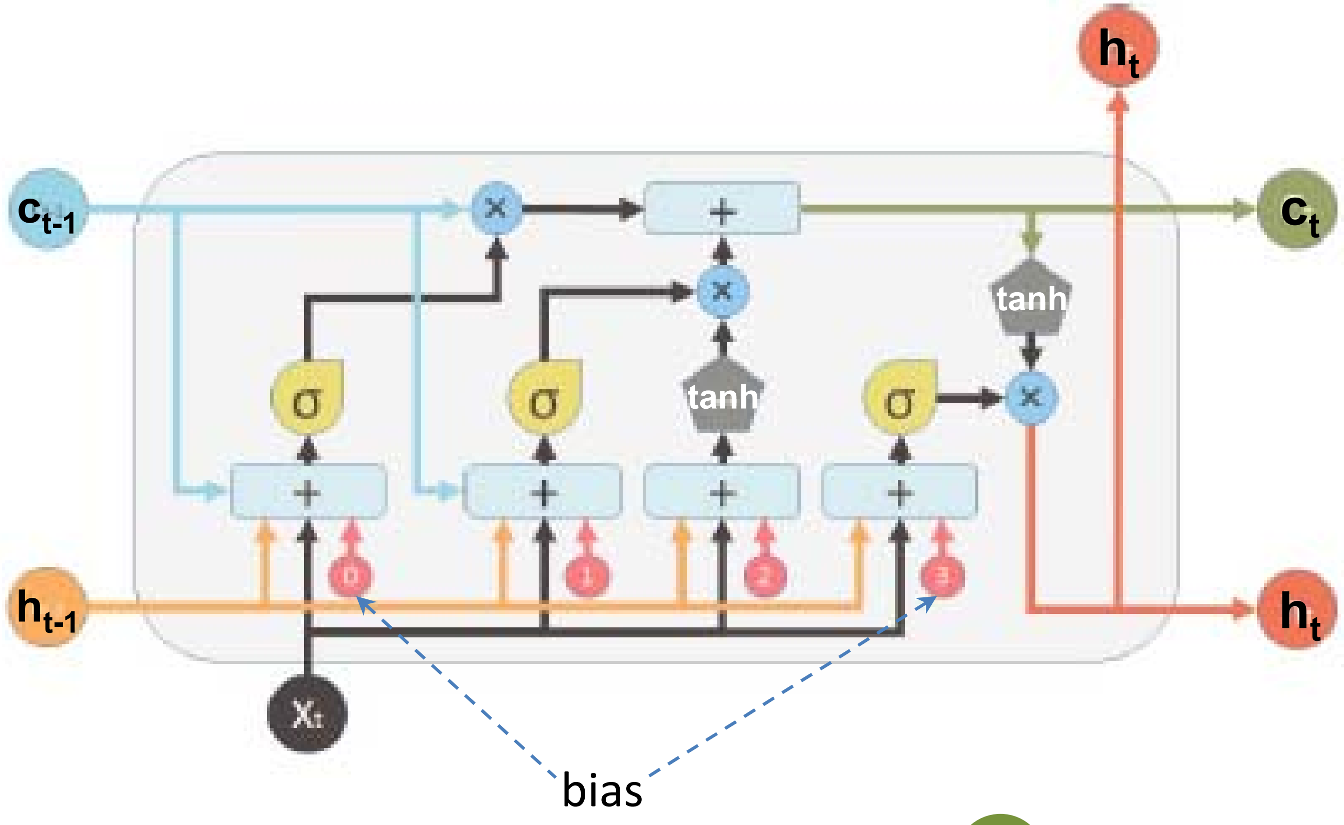
- explicit memory – **cell state vector** (on top of the *hidden* state)
- regulatory mechanism for information flow in and out – **gating unit**

# Long short-term memory (LSTM) network



Hochreiter S, Schmidhuber J. Long short-term memory. *Neural computation*. 1997 Nov 15;9(8):1735-80.

# Long short-term memory (LSTM) cell



$x_t$ input

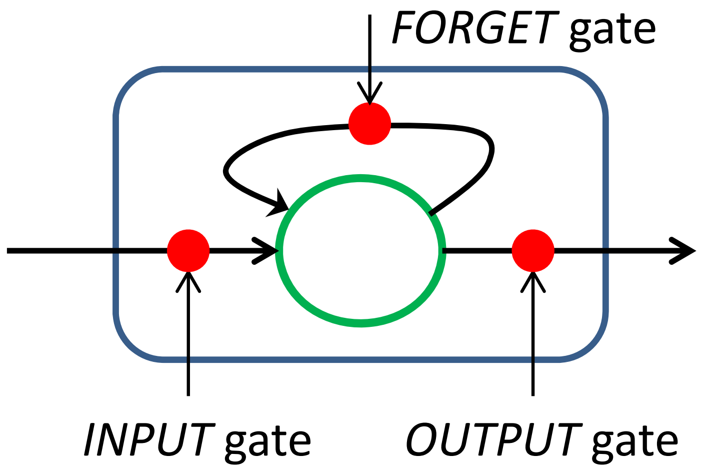
$c_{t-1}$ memory from previous cell

$h_{t-1}$ output from previous cell

● current memory

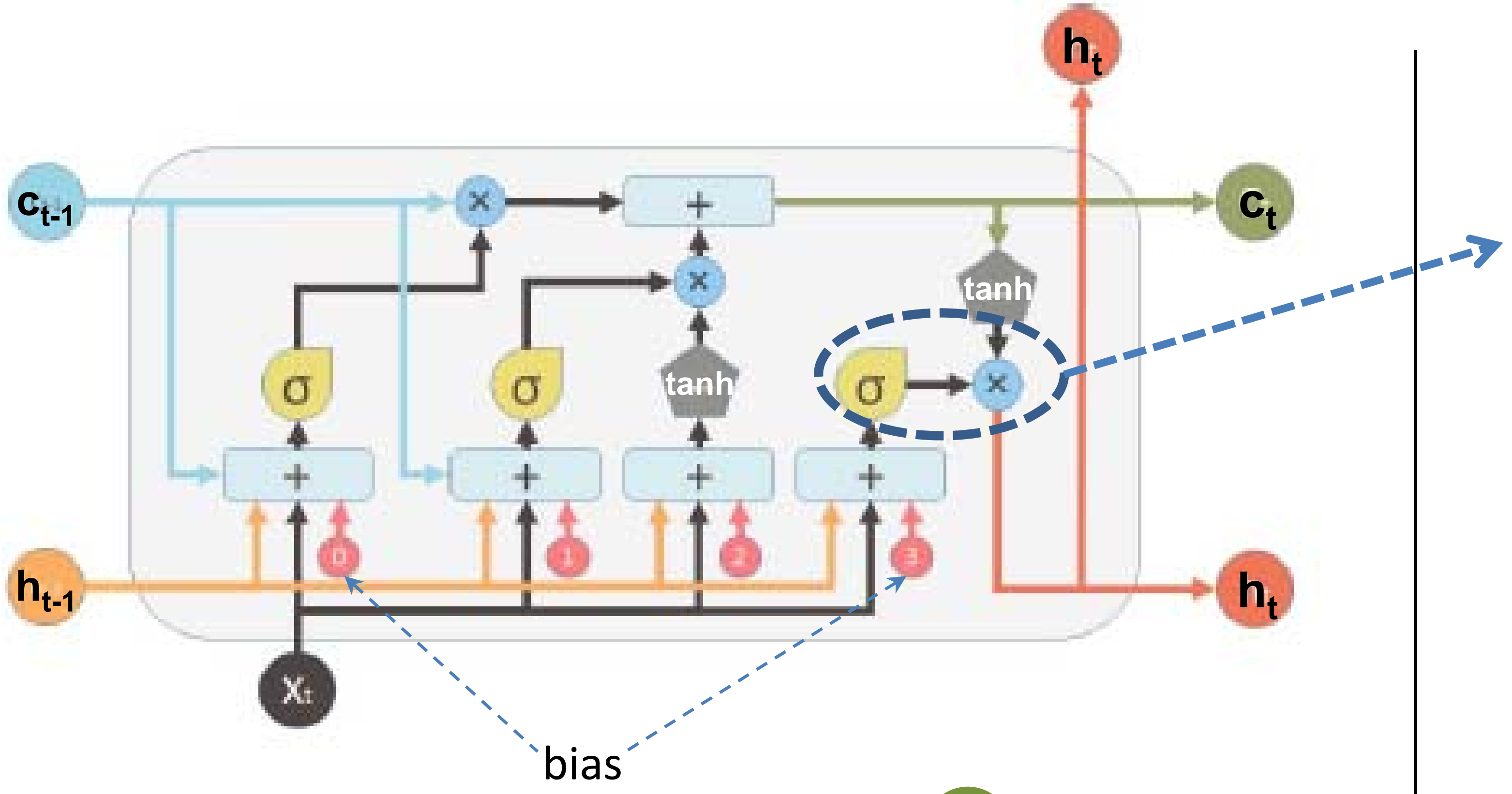
$h_t$ current output

+ concatenation



adapted from A. Sood and colah's blog

# Long short-term memory (LSTM) cell



$x_t$ input

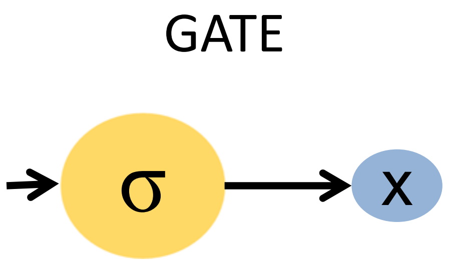
$c_{t-1}$ memory from previous cell

$h_{t-1}$ output from previous cell

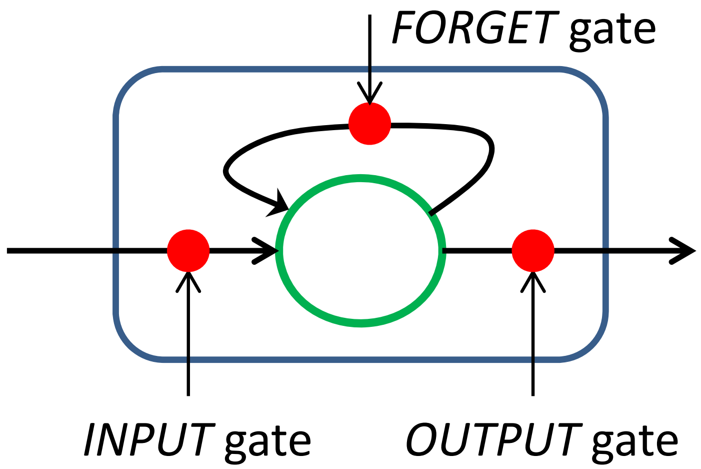
current memory

$h_t$ current output

+ concatenation

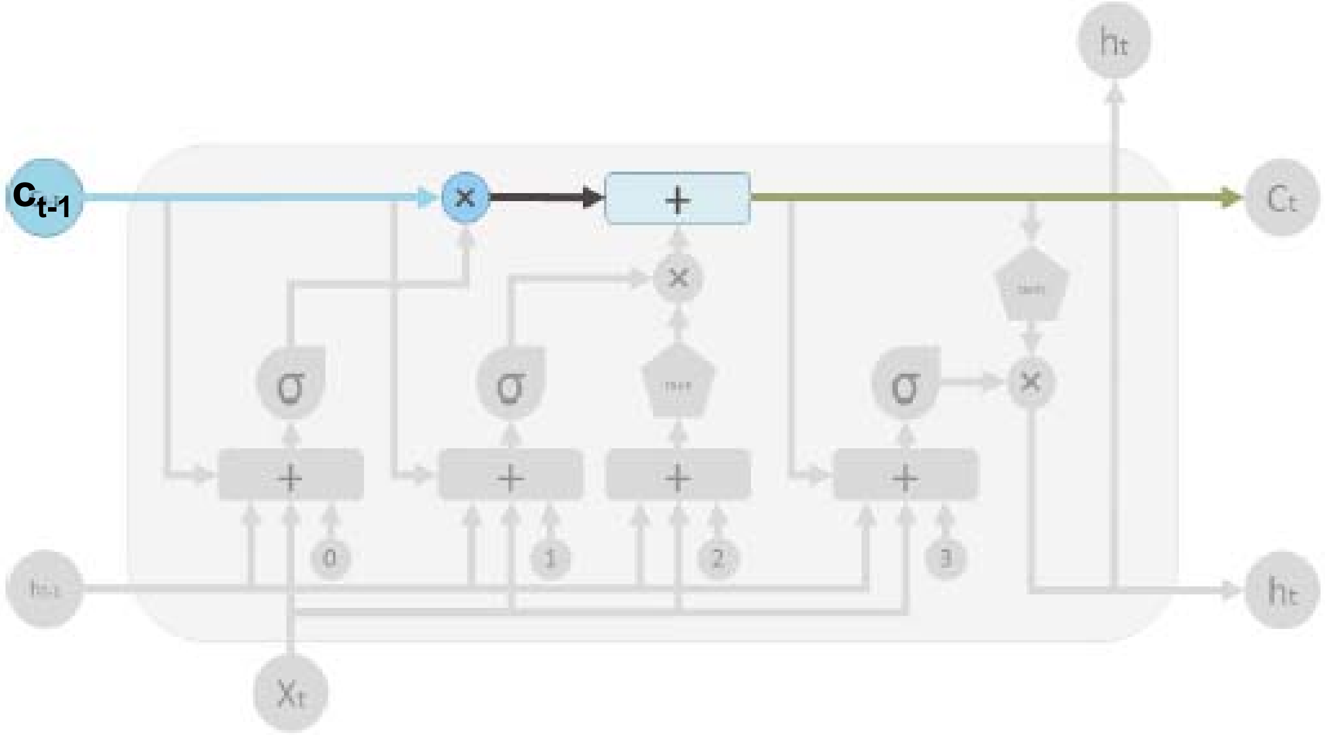


*sigmoid* followed by pointwise *multiplication*

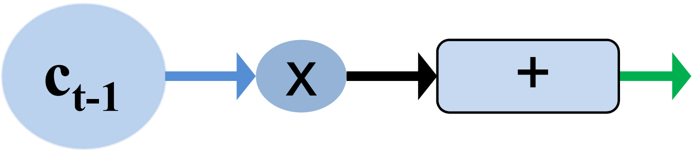


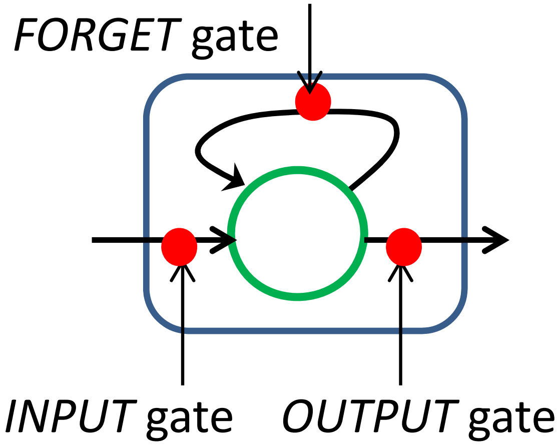
adapted from A. Sood and colah's blog

# Memory cell components – cell state vector



CELL STATE vector



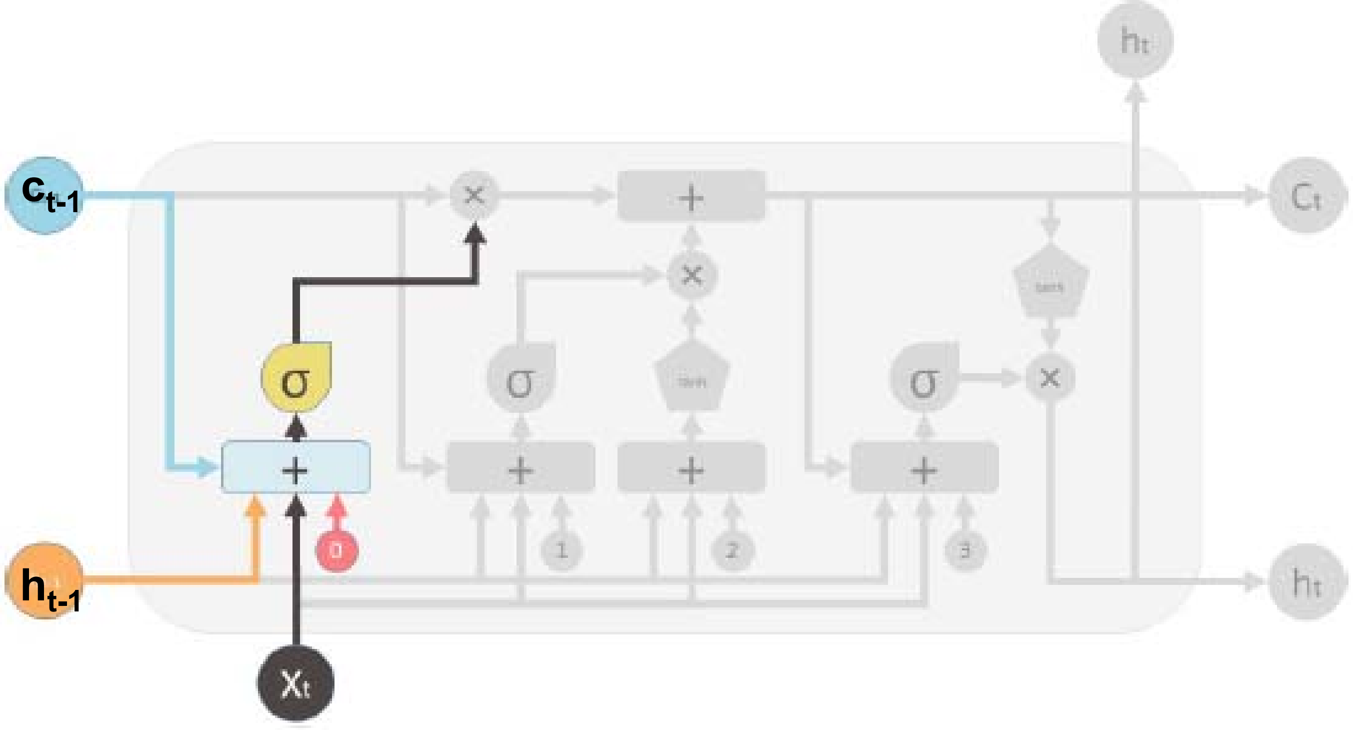


adapted from A. Sood and colah's blog

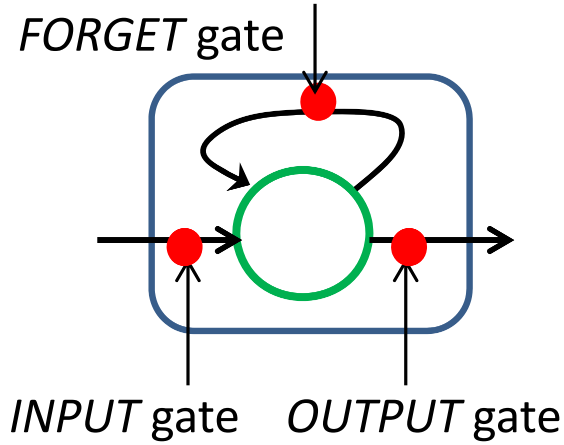
### Cell state vector

- represents memory
- it is changing as a result of new information (input gate) and forgetting (forget gate)

# Memory cell components – forget gate



$$f(t) = \sigma\left(\mathbf{W}_f\,[h(t-1), x(t)] + b_f\right)$$

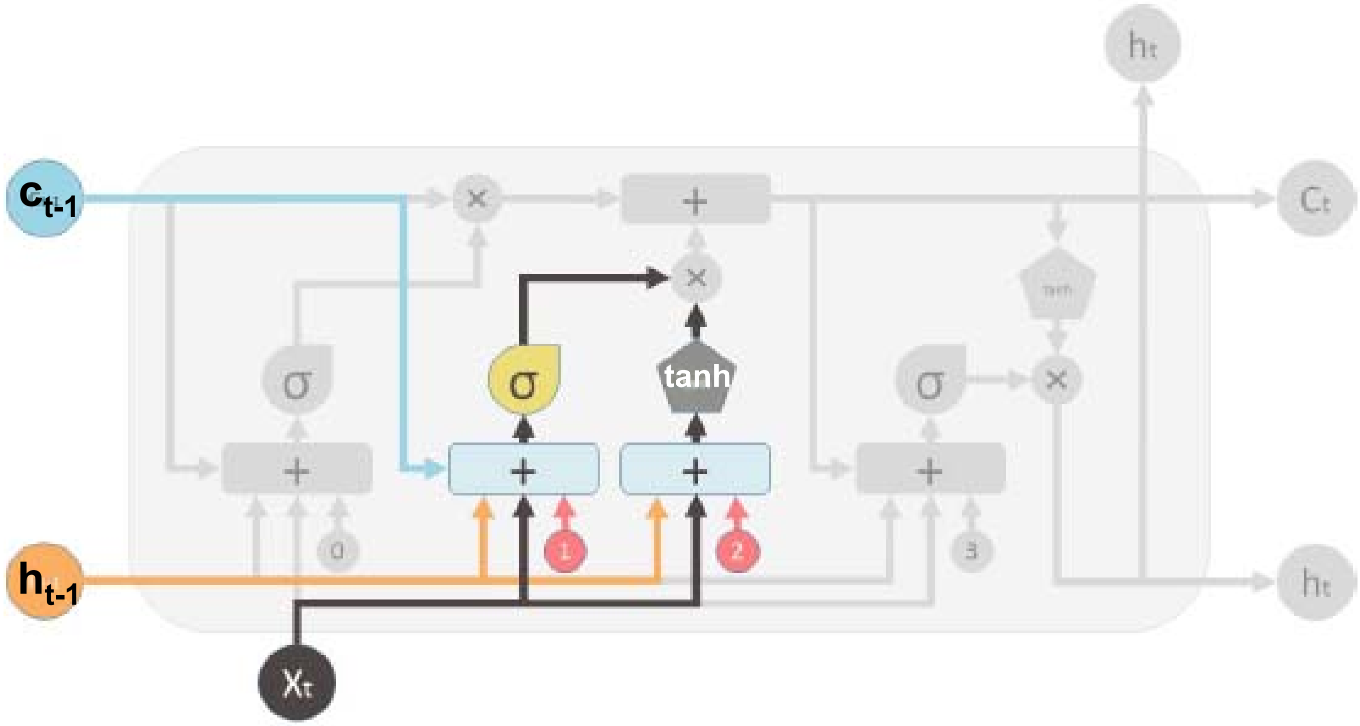


adapted from A. Sood and colah’s blog

**Forget gate**

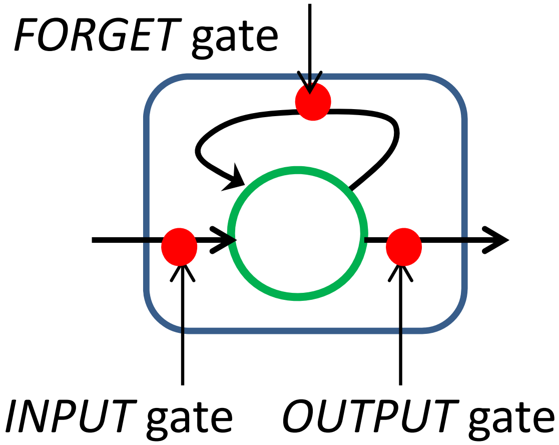
- controls what information should be forgotten (removed from memory)

# Memory cell components – input gate



$$i(t) = \sigma\left(\mathbf{W}_i\,[h(t-1), x(t)] + b_i\right)$$

$$\tilde{C}(t) = \tanh\left(\mathbf{W}_C\left[h(t-1), x(t)\right] + b_C\right)$$


*FORGET* gate

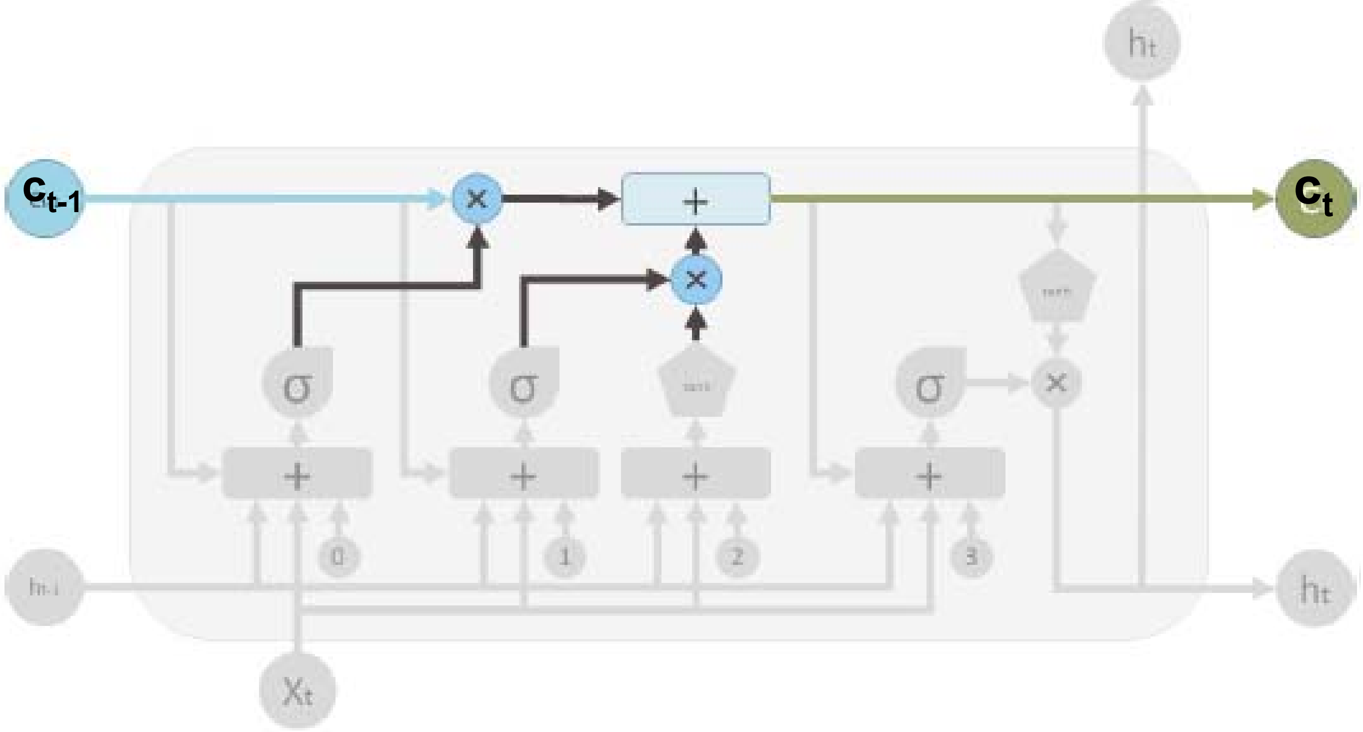
*INPUT* gate *OUTPUT* gate

adapted from A. Sood and colah's blog

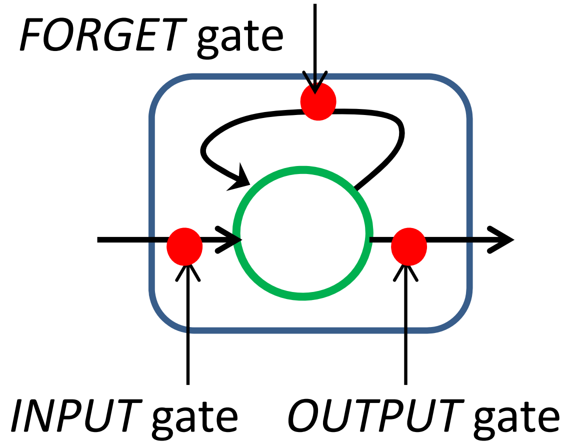
**Input gate**

- controls what information should be added to cell state from the current input

# Memory update



$$C(t) = f(t) \odot C(t-1) + i(t) \odot \tilde{C}(t)$$

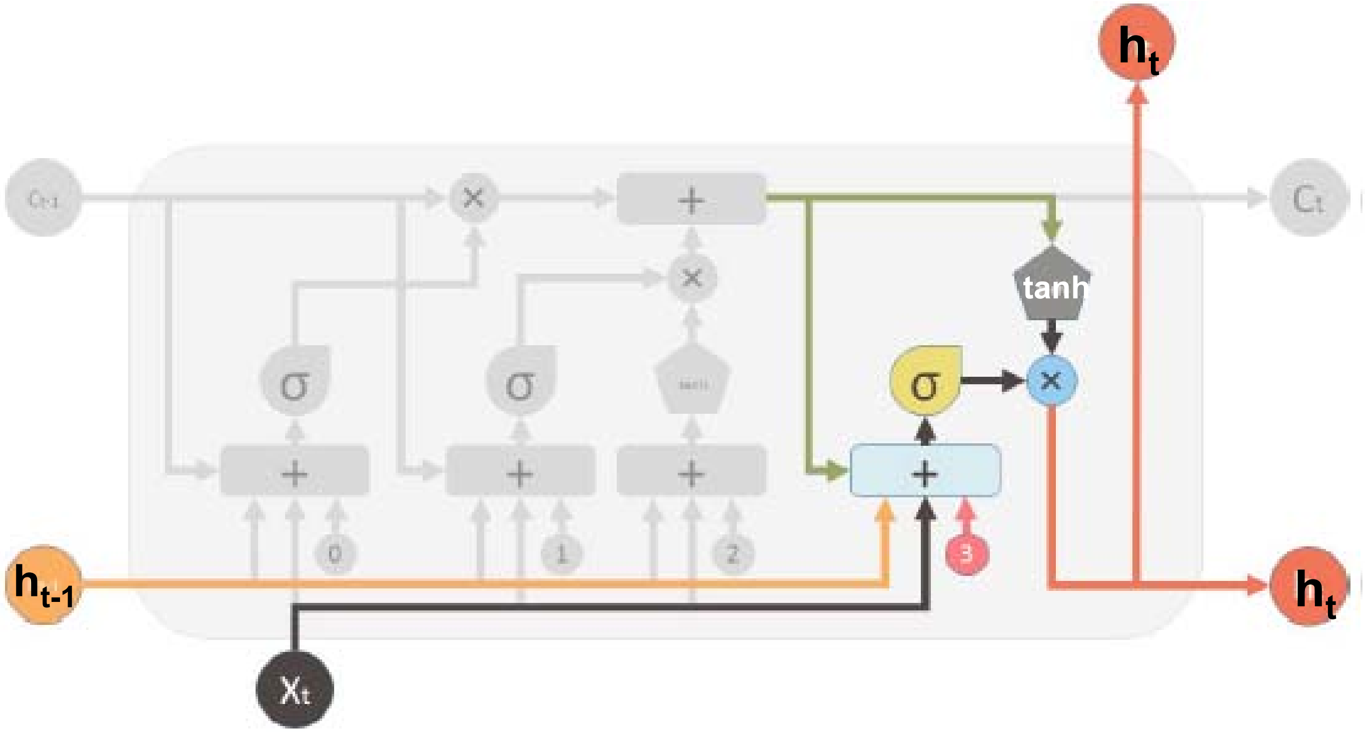


adapted from A. Sood and colah's blog

### Updating memory

- cell state vector aggregates old memory, gated by forget gate, and a new memory, filtered by the input gate

# Memory cell components – output gate

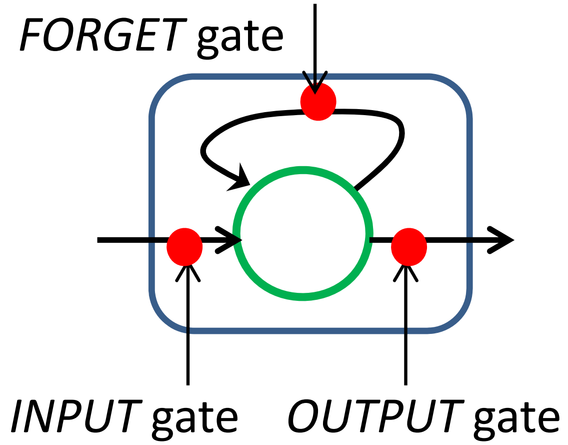


$$o(t) = \sigma\left(\mathbf{W}_o[h(t-1), x(t)] + b_o\right)$$

$$h(t) = o(t) \odot \tanh\left(C(t)\right)$$

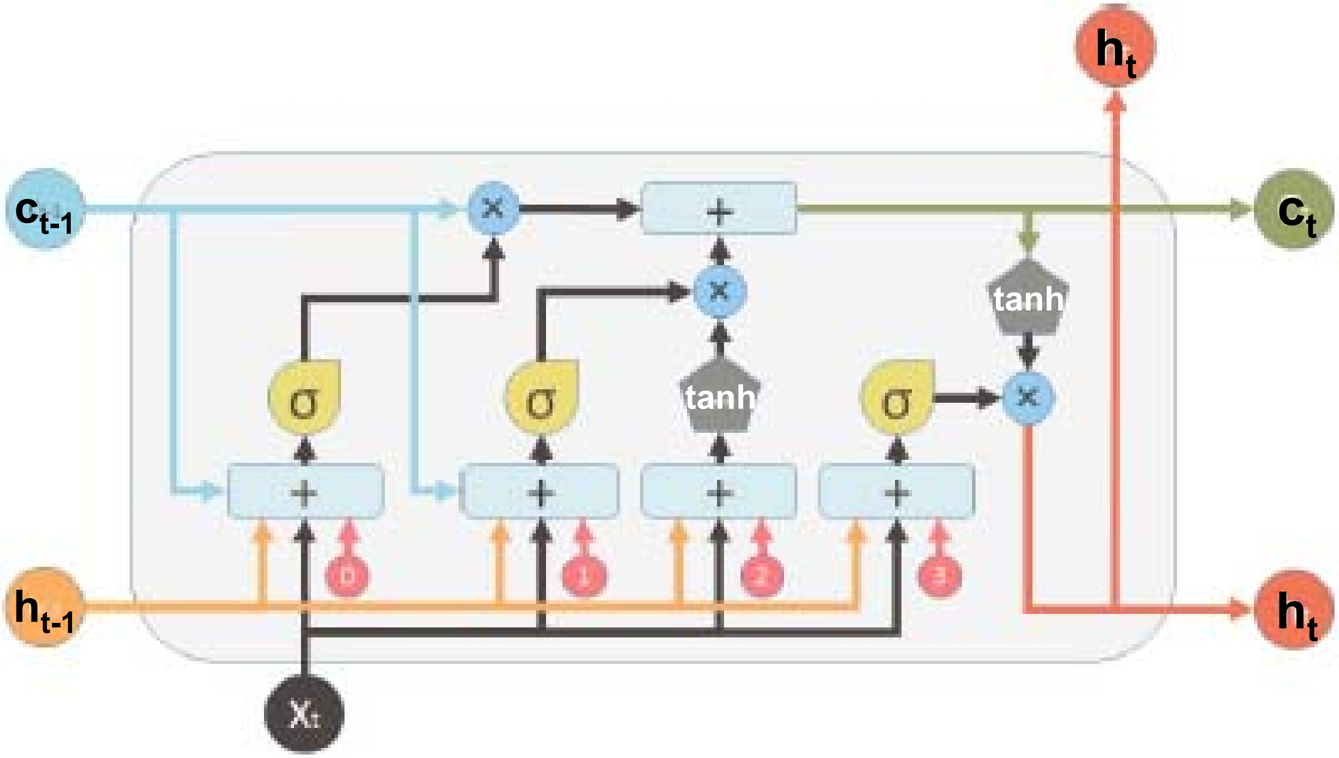
## Output gate

- controls what information is sent to the output



adapted from A. Sood and colah's blog

# LSTM memory cell summary



$$f(t) = \sigma\left(\mathbf{W}_f\,[h(t-1), x(t)] + b_f\right)$$

$$i(t) = \sigma\left(\mathbf{W}_i\,[h(t-1), x(t)] + b_i\right)$$

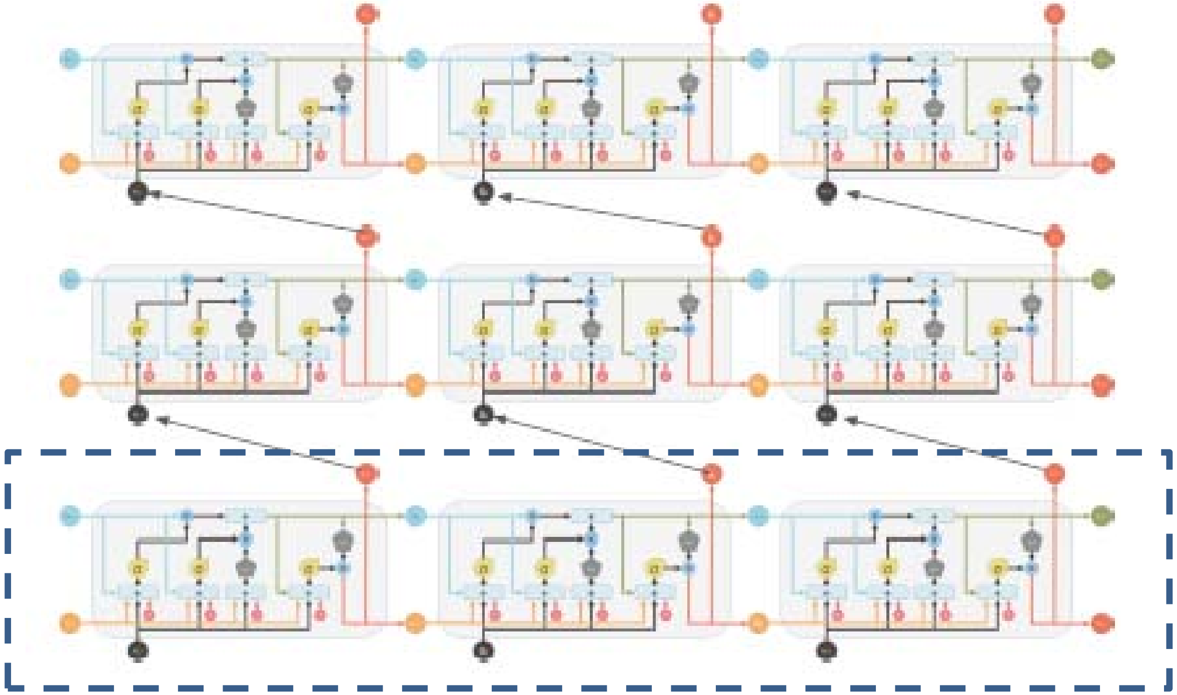
$$o(t) = \sigma\left(\mathbf{W}_o\,[h(t-1), x(t)] + b_o\right)$$

$$\tilde{C}(t) = \tanh\left(\mathbf{W}_C\left[h(t-1), x(t)\right] + b_C\right)$$

$$C(t) = f(t) \odot C(t-1) + i(t) \odot \tilde{C}(t)$$

$$h(t) = o(t) \odot \tanh\left(C(t)\right)$$

# Deep LSTM

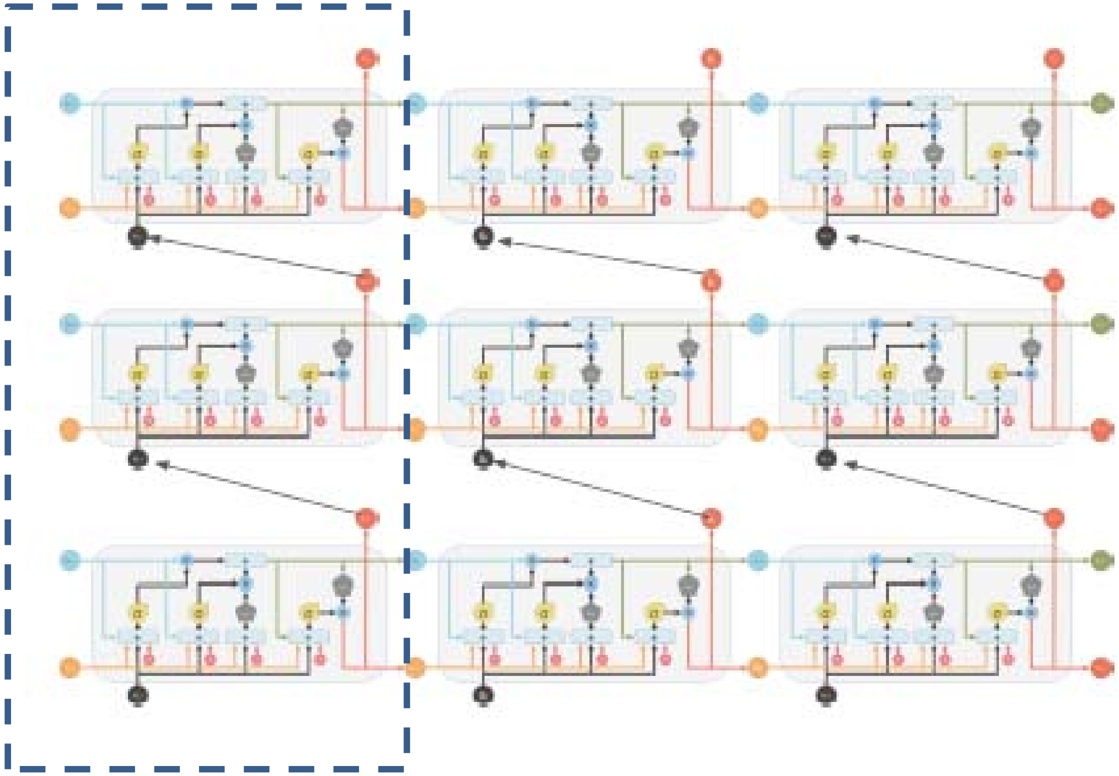


temporal unfolding

# Deep LSTM

deep stacking

output sequence of one layer constitutes the input sequence to another layer



**Why do we go deep?**

- has the potential to perform better at handling temporal information at wide varying scales
- requires however many more parameters to be learnt

adapted from A. Sood

# Video example of text understanding with LSTM

*https://youtu.be/mLxsbWAYIpw*