Machine Learning (part II)

Recurrent Neural Networks

Angelo Ciaramella

Recurrent Neural Networks

- RNNs
  - family of neural networks for processing sequential data
  - specialized for processing a sequence of values

$$x^{(1)}, \ldots, x^{(\tau)}$$

  - early ideas found in machine learning and statistical models of the 1980s
    - sharing parameters across different parts of a model

Recurrent Neural Networks

- Related idea
  - use of convolution across a 1-D temporal sequence
    - time-delay neural networks
- RNNs
  - minibatches of sequences
  - may also be applied in two dimensions across spatial data such as images

Adaptive filters

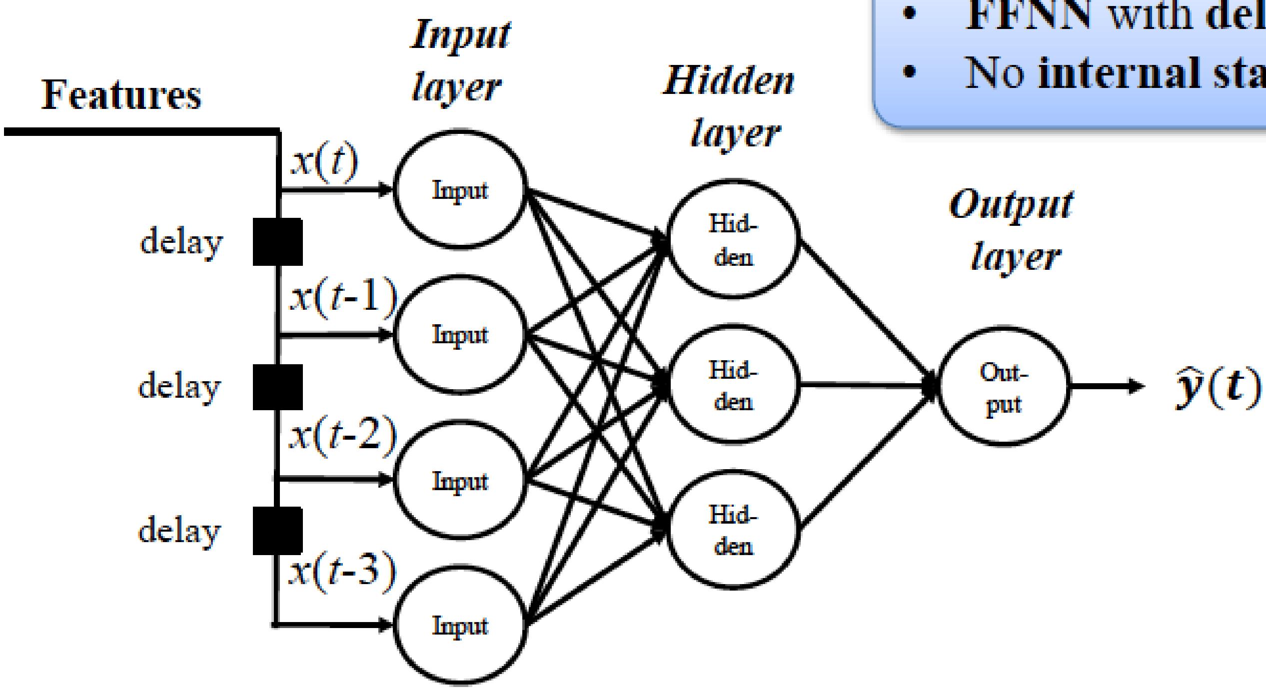
- Adaptive filter
  - The parameters are estimated
    - learning algorithm
  - An error function is used
  - e.g., Linear Artificial Neural Network (Adaline)

Adaptive filters

- Hospital
  - ECG (electrocardiogram) corrupted by noise at 50 Hz (electricity)
  - The current can vary between 47 Hz and 53 Hz
  - A filter for the elimination of static noise at 50 Hz could give errors
  - An adaptive filter can learn from the current shape of noise

- Helicopter
  - Pilot speaking with noise from rotating propeller
  - The noise has not a spectrum well defined
  - An adaptive filter learns the shape of the noise
  - The noise can be subtracted from the signal for only the pilot's voice

Adaline

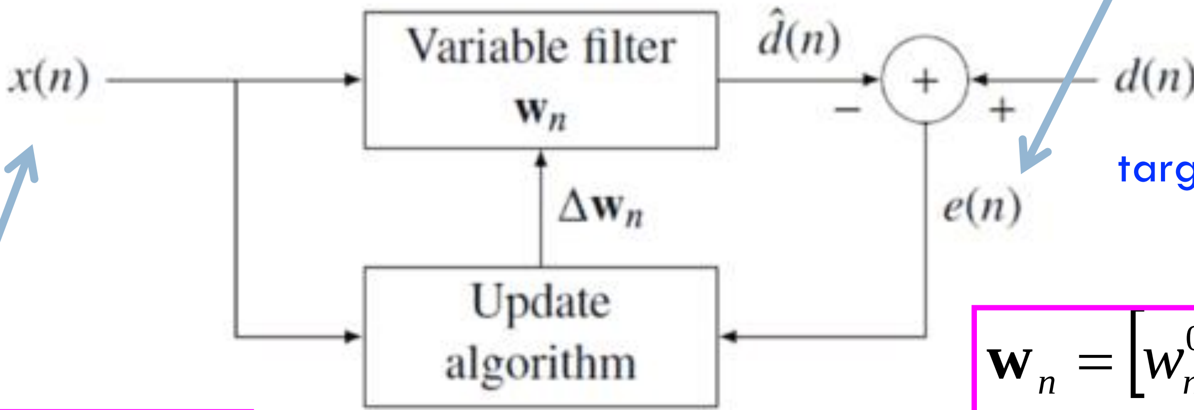


- **FFNN** with **delayed** inputs
- No **internal state**

Adaptive filters

$$e(n) = d(n) - \hat{d}(n)$$

input



target signal

$$\mathbf{w}_n = \left[w_n^0, w_n^1, \ldots, w_n^p\right]$$

$$x(n) = d(n) + v(n)$$

noise

FIR structure

$$\tilde{d}(n) = \mathbf{w}_n \cdot \mathbf{x}(n)$$

$$\mathbf{w}_{n+1} = \mathbf{w}_n + \Delta\mathbf{w}_n$$

Learning algorithm (e.g., LMS and descent gradient)

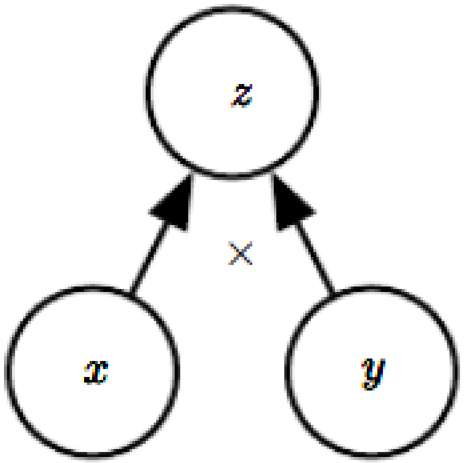
$$\mathbf{x}(n) = \left[x(n), x(n-1), \ldots, x(n-p)\right]$$

source signal

Computational graphs

- Related idea
  - formalize the structure of a set of computations
  - introduce the idea of an operation
    - an operation is a simple function of one or more variables

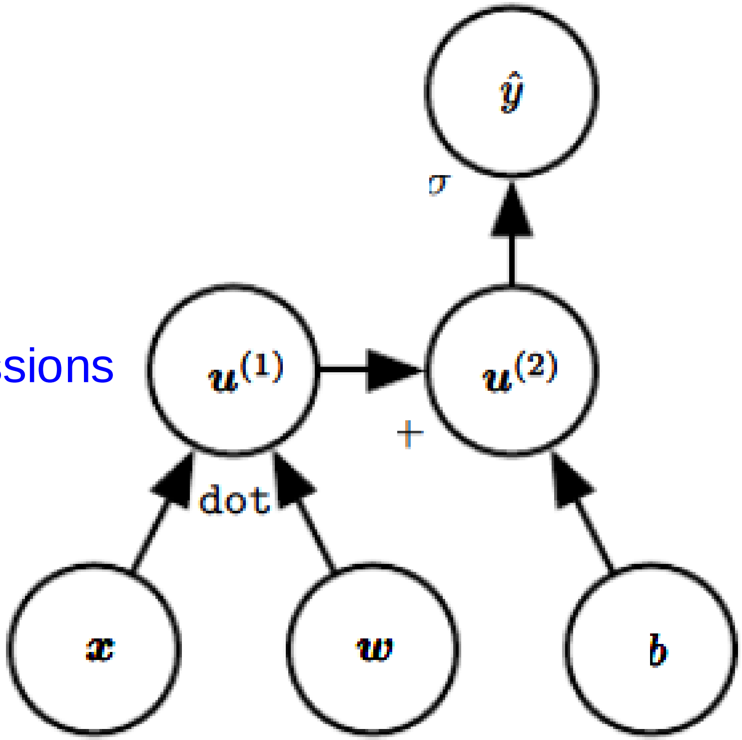
Computational graphs



Graph using the × operation to compute z = xy

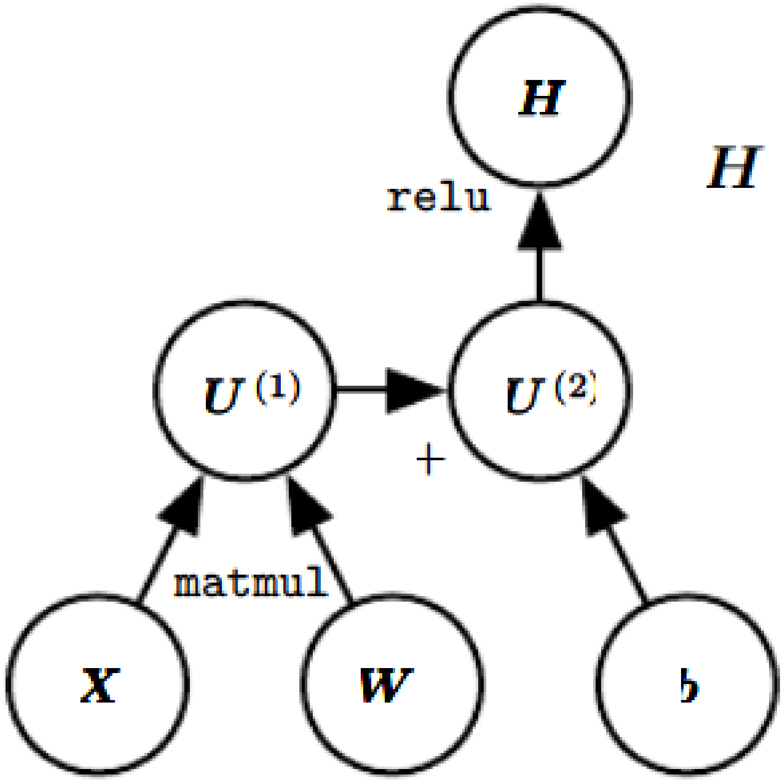
Computational graphs

intermediate expressions



logistic regression prediction $\hat{y} = \sigma\left(x^\top w + b\right)$

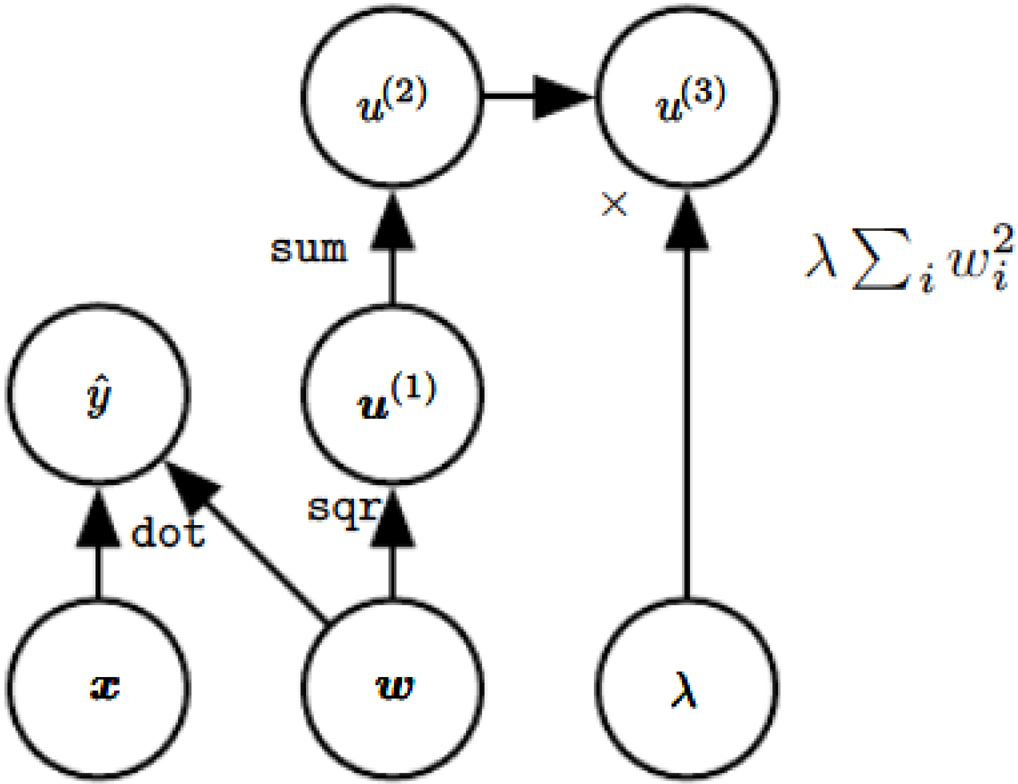
Computational graphs



$$\boldsymbol{H} = \max\{0, \boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}\}$$

Minibatch of inputs X

Computational graphs

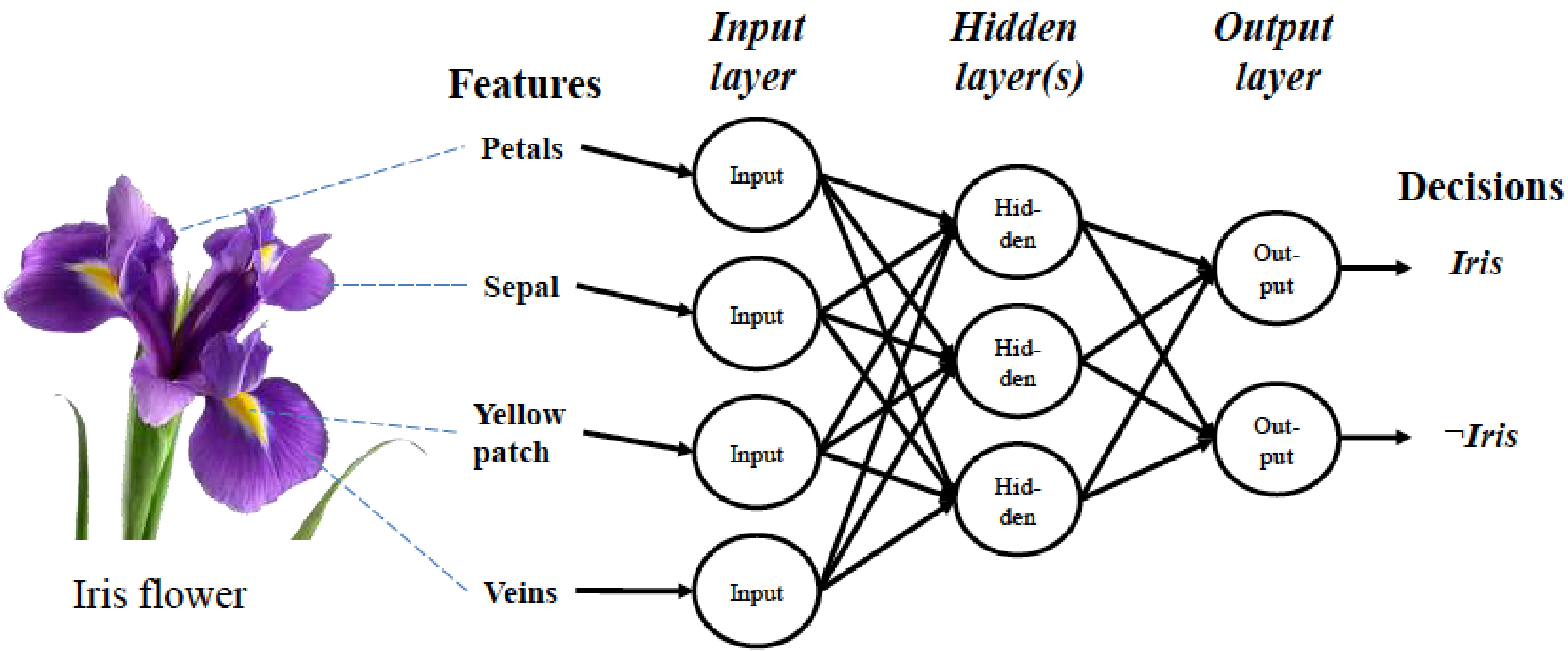


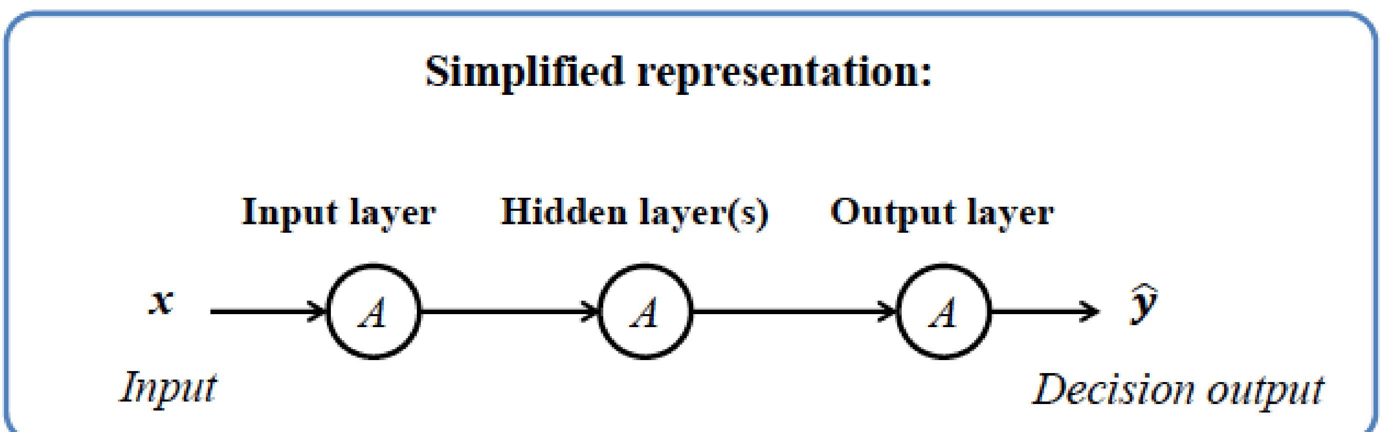
More than one operation of a linear regression model

RNNs

- Artificial Neural Networks
  - exhibit temporal dynamic behavior
  - can use their internal state (memory) to process sequences of inputs
  - models sequences
    - Time series
    - Natural Language
    - Speech
    - Convert non-sequences to sequences, eg: feed an image as a sequence of pixels!

Feed-forward NN





Temporal dipendencies

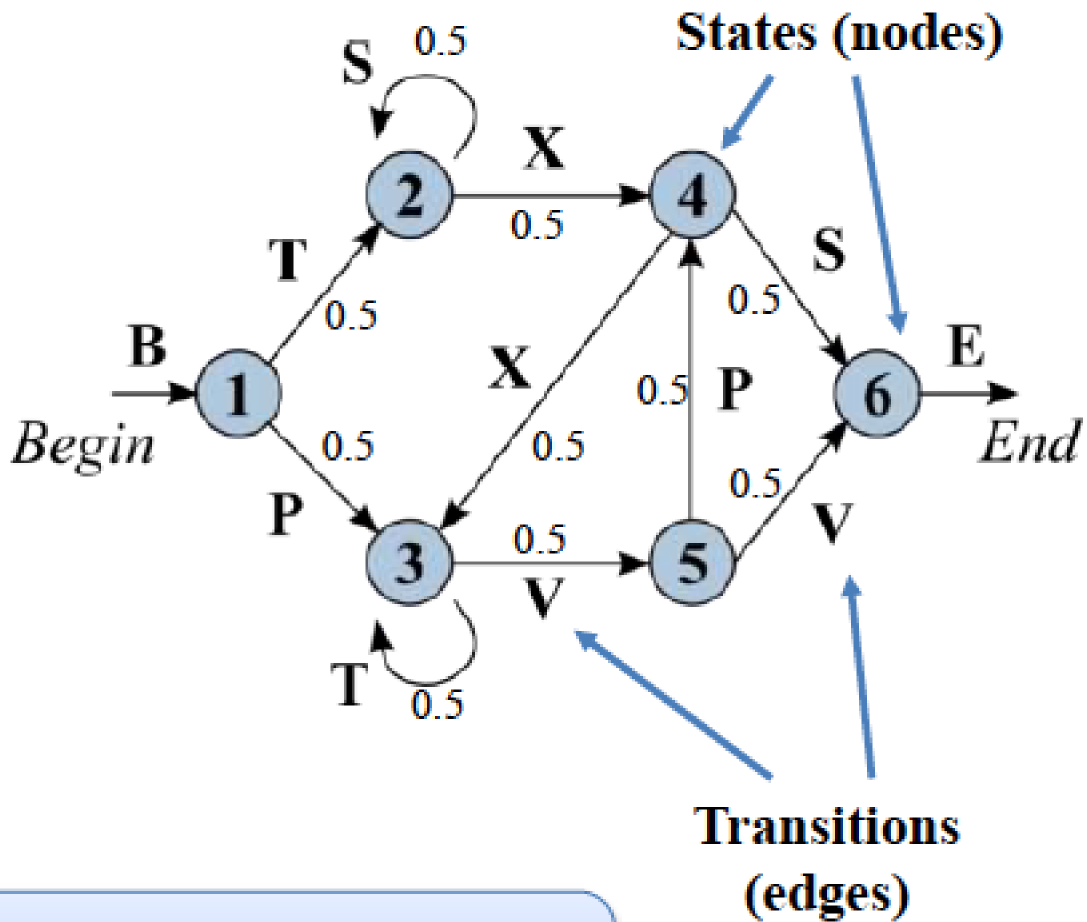
Frame 0	Frame 1	Frame 2	Frame 3	Frame 4
Stem: seen Petals: hidden	Stem: seen Petals: hidden	Stem: seen Petals: partial	Stem: partial Petals: partial	Stem: hidden Petals: seen

P(*Iris*): 0.1 *P*(¬*Iris*): 0.9	*P*(*Iris*): 0.11 *P*(¬*Iris*): 0.89	*P*(*Iris*): 0.2 *P*(¬*Iris*): 0.8	*P*(*Iris*): 0.45 *P*(¬*Iris*): 0.55	***P*(*Iris*): 0.9** *P*(¬*Iris*): 0.1

Decision on sequence of observations

Reber Grammar

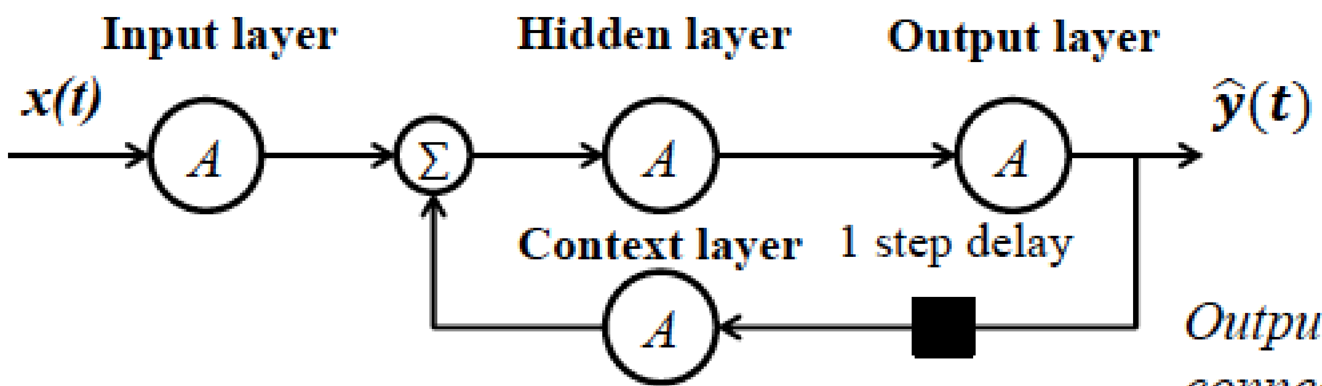
Problem that can not be solved without memory



Transitions have equal probabilities:
$\boldsymbol{P}(1\rightarrow 2) = \boldsymbol{P}(1\rightarrow 3) = \mathbf{0.5}$

Jordan's sequential network

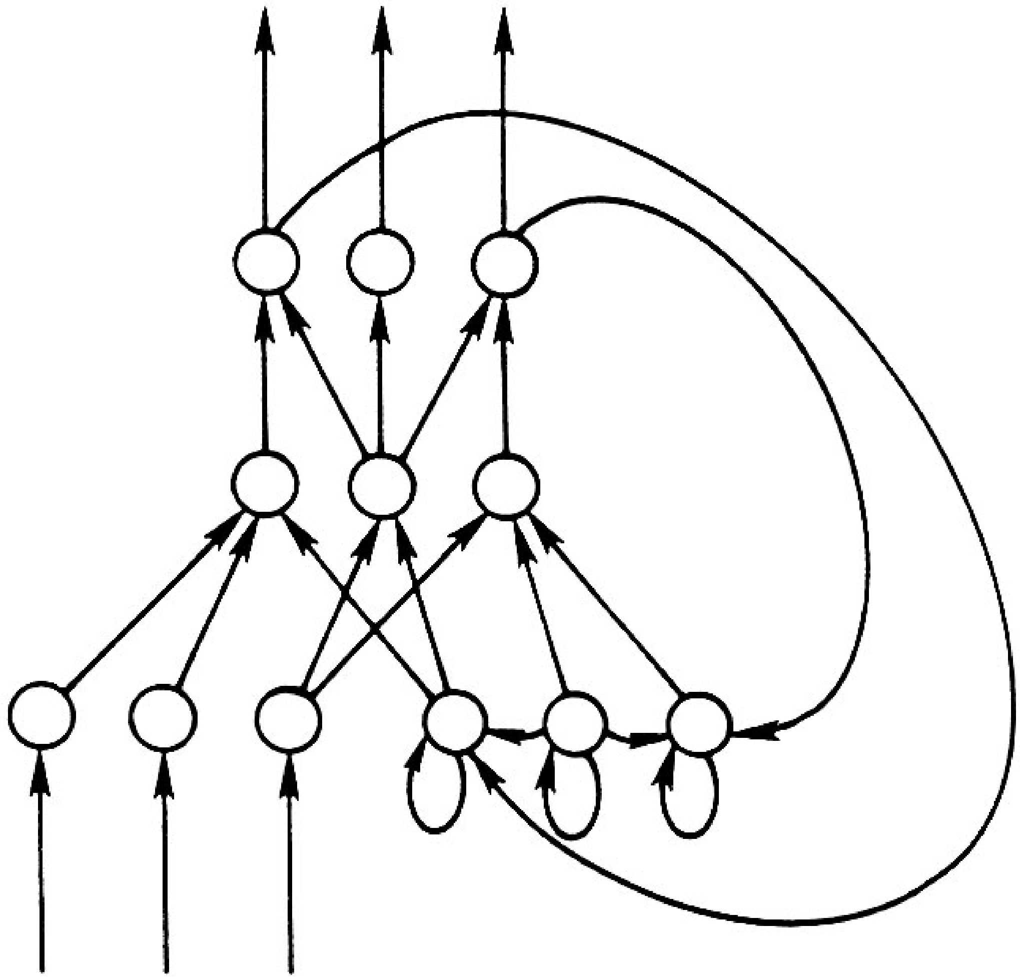
Limited short-term memory



Output-to hidden connections

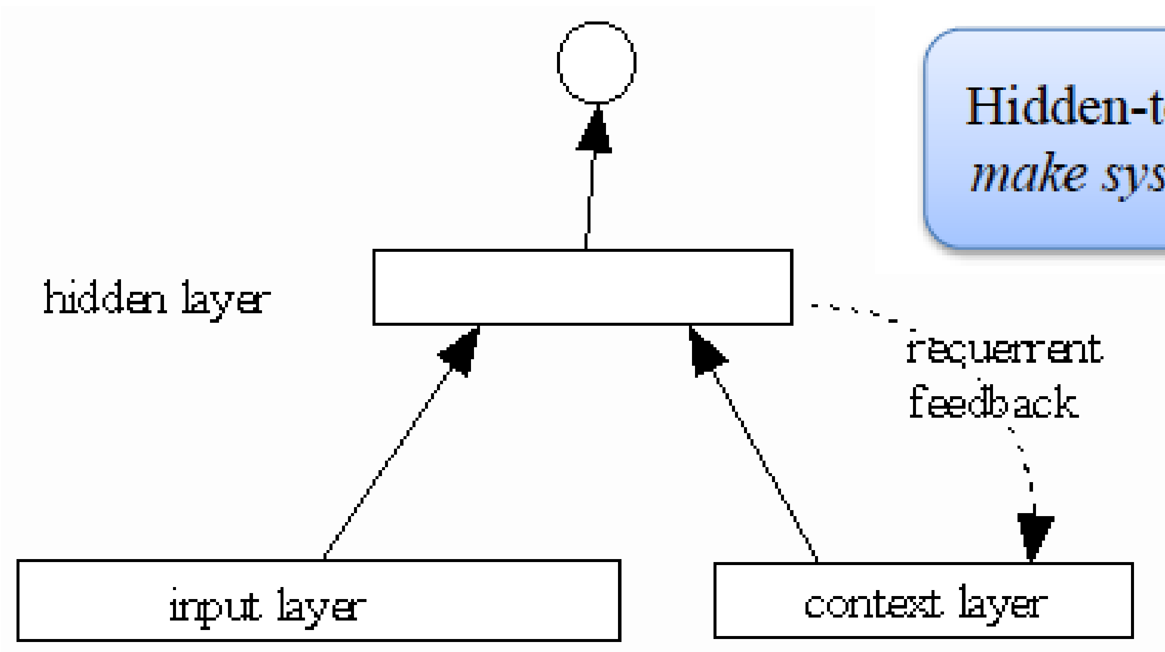
M.I. Jordan NN (1986).

Jordan's sequential network

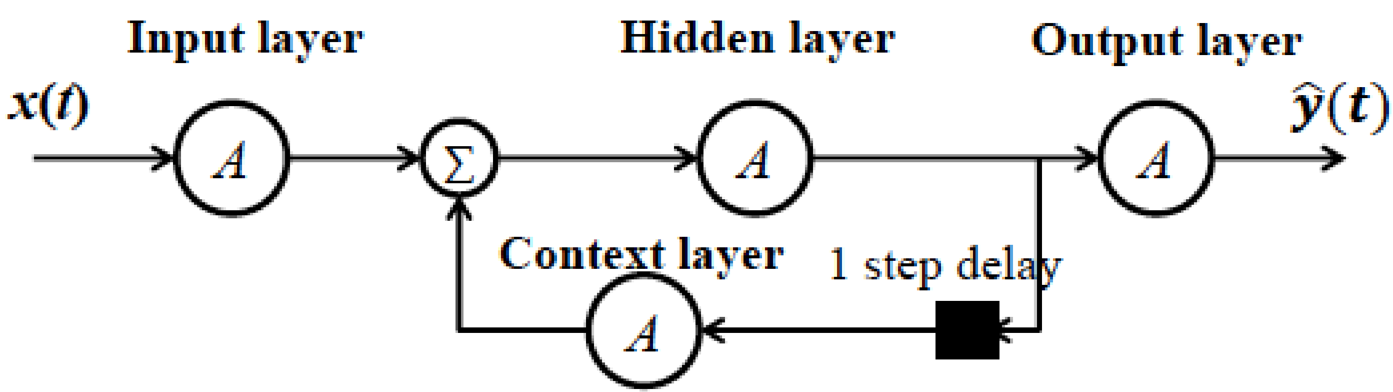


Jordan NN has been applied to categorize a class of English syllables.

Simple recurrent network

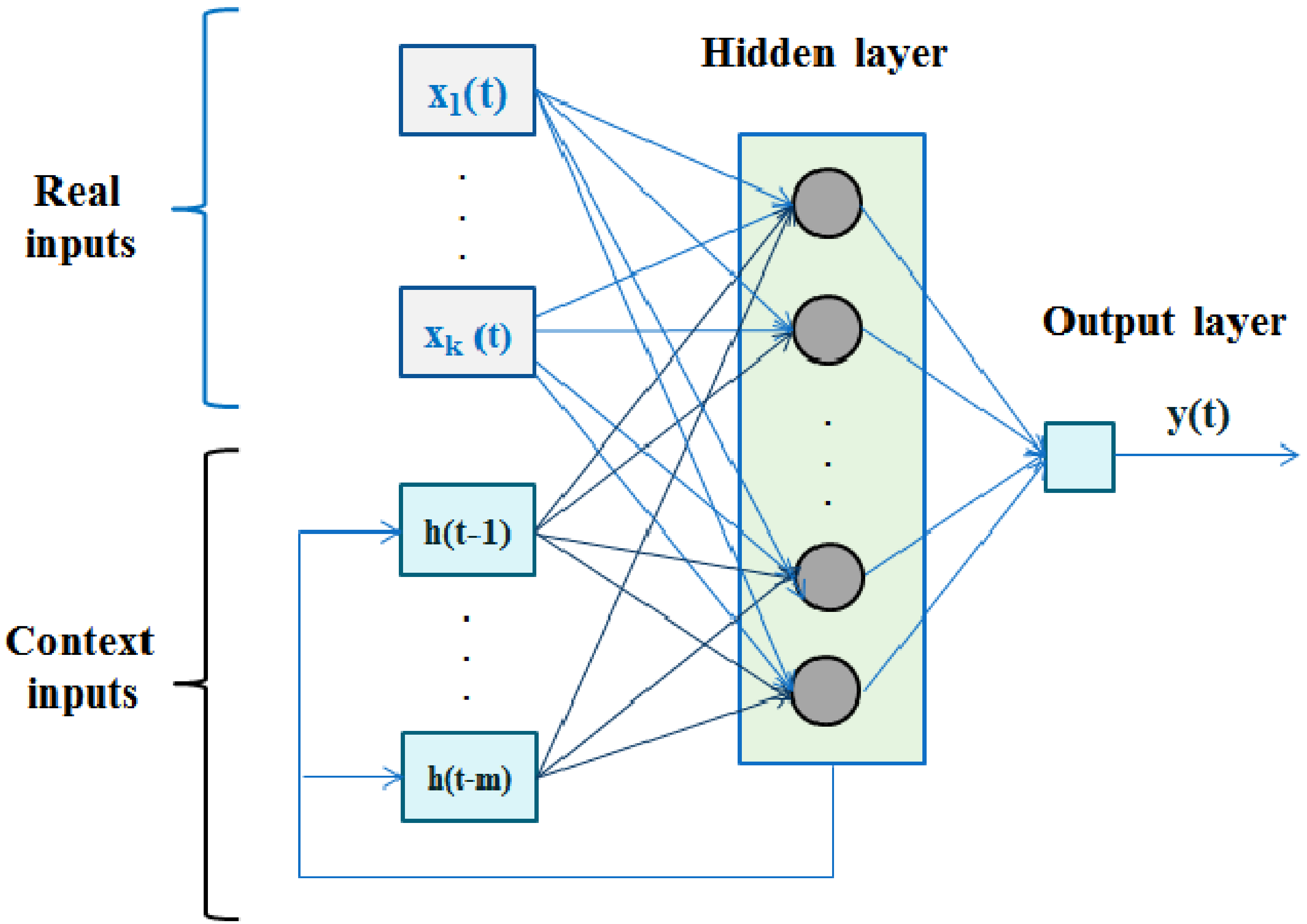


Hidden-to hidden connections
make system Turing-complete



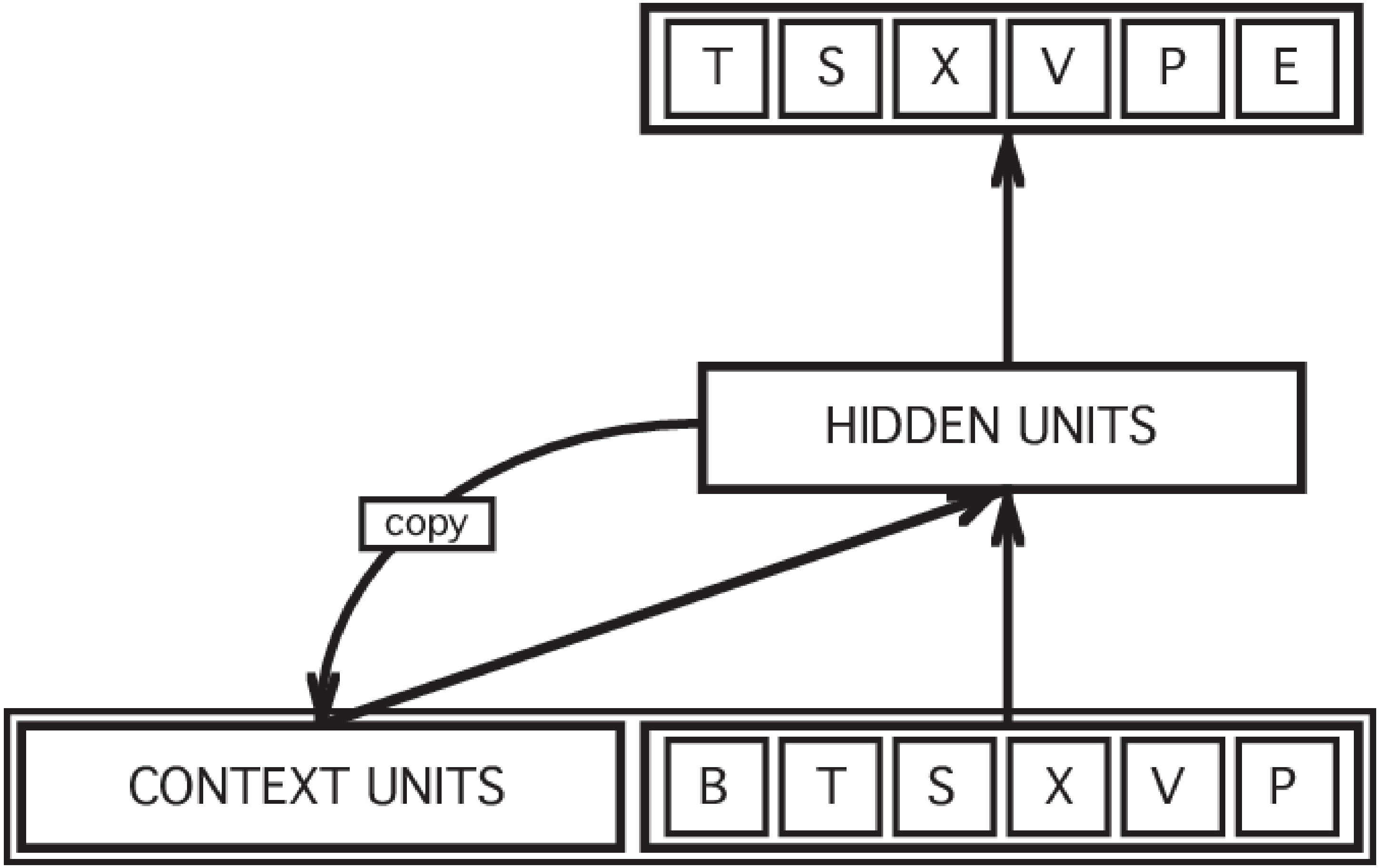
Elman RNN (1990).

Simple recurrent network



Elman RNN. Basic RNN structure called “Vanilla” RNN

Simple recurrent network



Elman SRN. A total of 60,000 randomly generated strings are used for training.

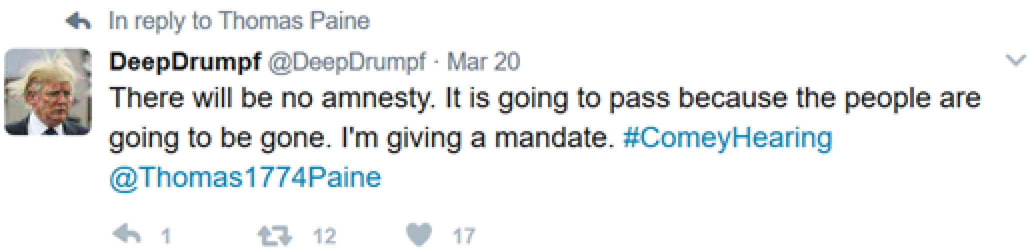
Applications of RNNs

A person riding a motorcycle on a dirt road.

Image Captioning

VIOLA:
Why, Salisbury must find his flesh and thought
That which I am not aps, not a man and in fire,
To show the reining of the raven and the wars
To grace my hand reproach within, and not a fair are hand,
That Caesar and my goodly father's world;

Write like Shakespeare



.. and Trump

RNN Generated Music
RNN Generated Eminem rapper

... and more!

Twitterbot
I'm a Neural Network trained on Trump's transcripts. Priming text in [ ]s. Donate (http://www.gofundme.com/deepdrumpf) to interact! Created by @hayesbh.

Recurrent Neural Networks

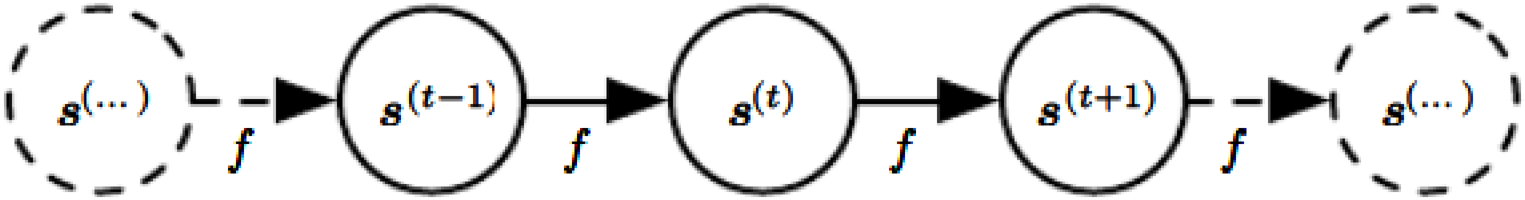
- Dynamical system (reccurrent expression)

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}; \boldsymbol{\theta})$$

State of the system

- Example

$$\begin{aligned}\boldsymbol{s}^{(3)} &= f(\boldsymbol{s}^{(2)}; \boldsymbol{\theta}) \\ &= f(f(\boldsymbol{s}^{(1)}; \boldsymbol{\theta}); \boldsymbol{\theta})\end{aligned}$$



Unfolded computational graph

Recurrent Neural Networks

- Dynamical system driven by an external signal

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta})$$

- RNNs

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

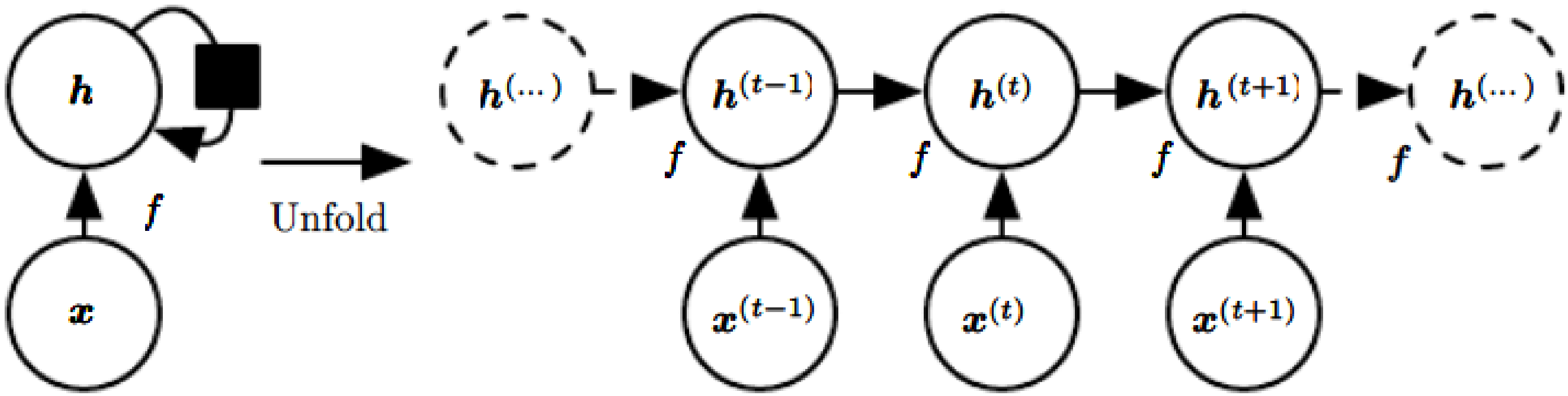
state of the hidden units of the network

Recurrent Neural Networks

A RNN with no outputs

Circuit diagram

delay of 1 time step



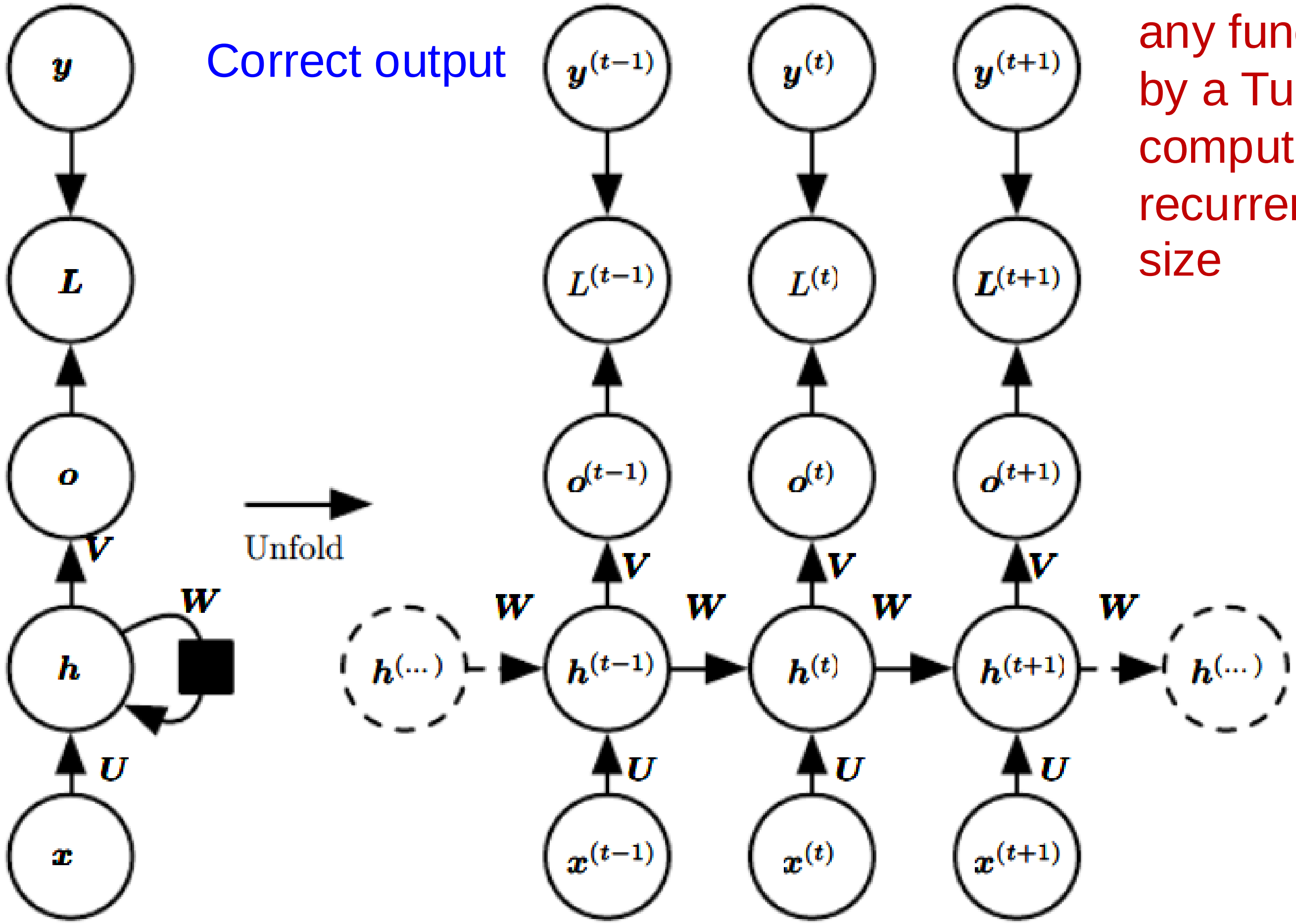
This recurrent network just processes information from the input x by incorporating it into the state h that is passed forward through time

Recurrent Neural Networks

- Representation of the unfolded recurrence

$$
\begin{aligned}
\boldsymbol{h}^{(t)} &= g^{(t)}\left(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}\right) \\
&= f\left(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}\right)
\end{aligned}
$$

Recurrent Neural Networks

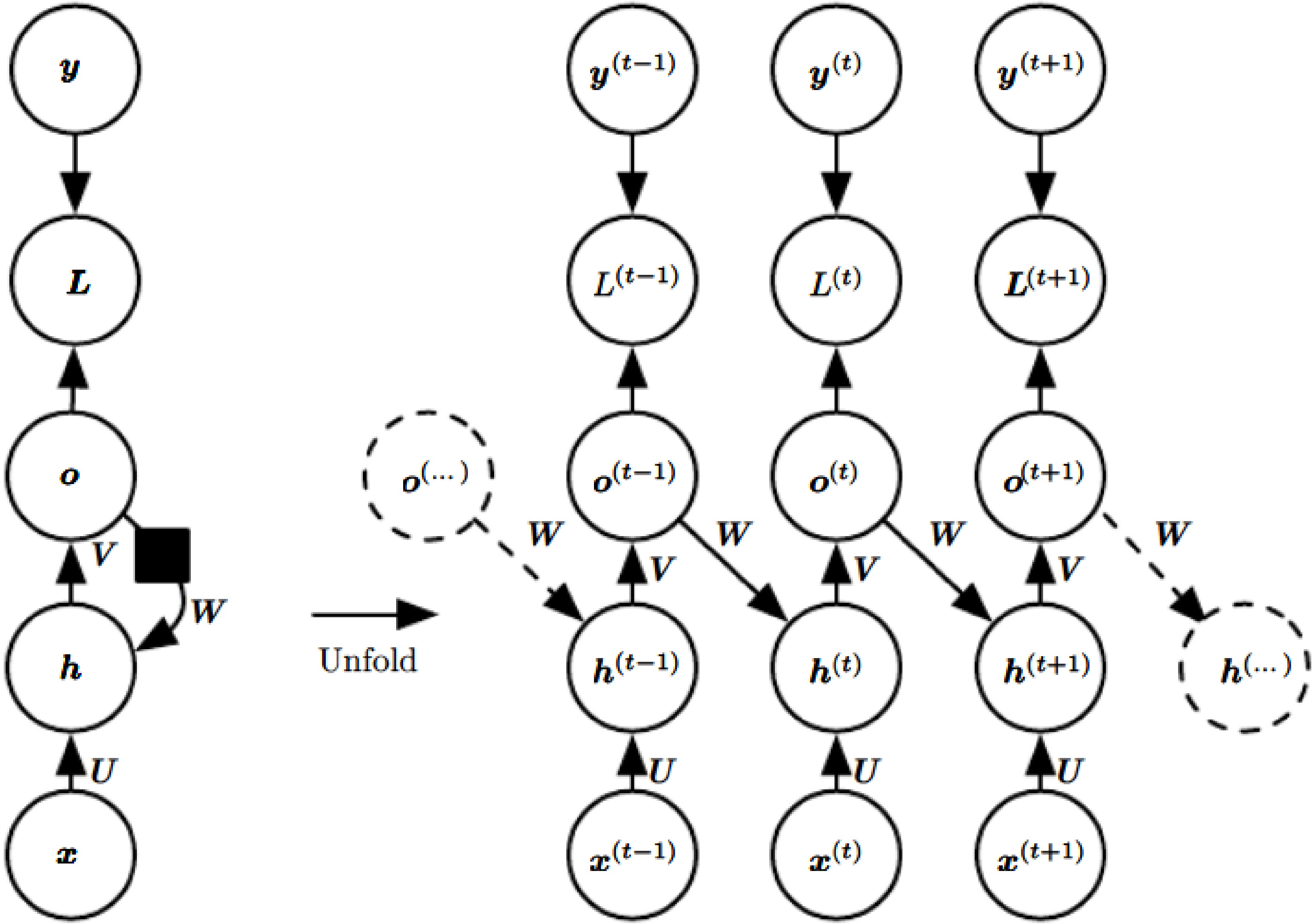


Correct output

universal –
any function computable by a Turing machine can be computed by such a recurrent network of a finite size

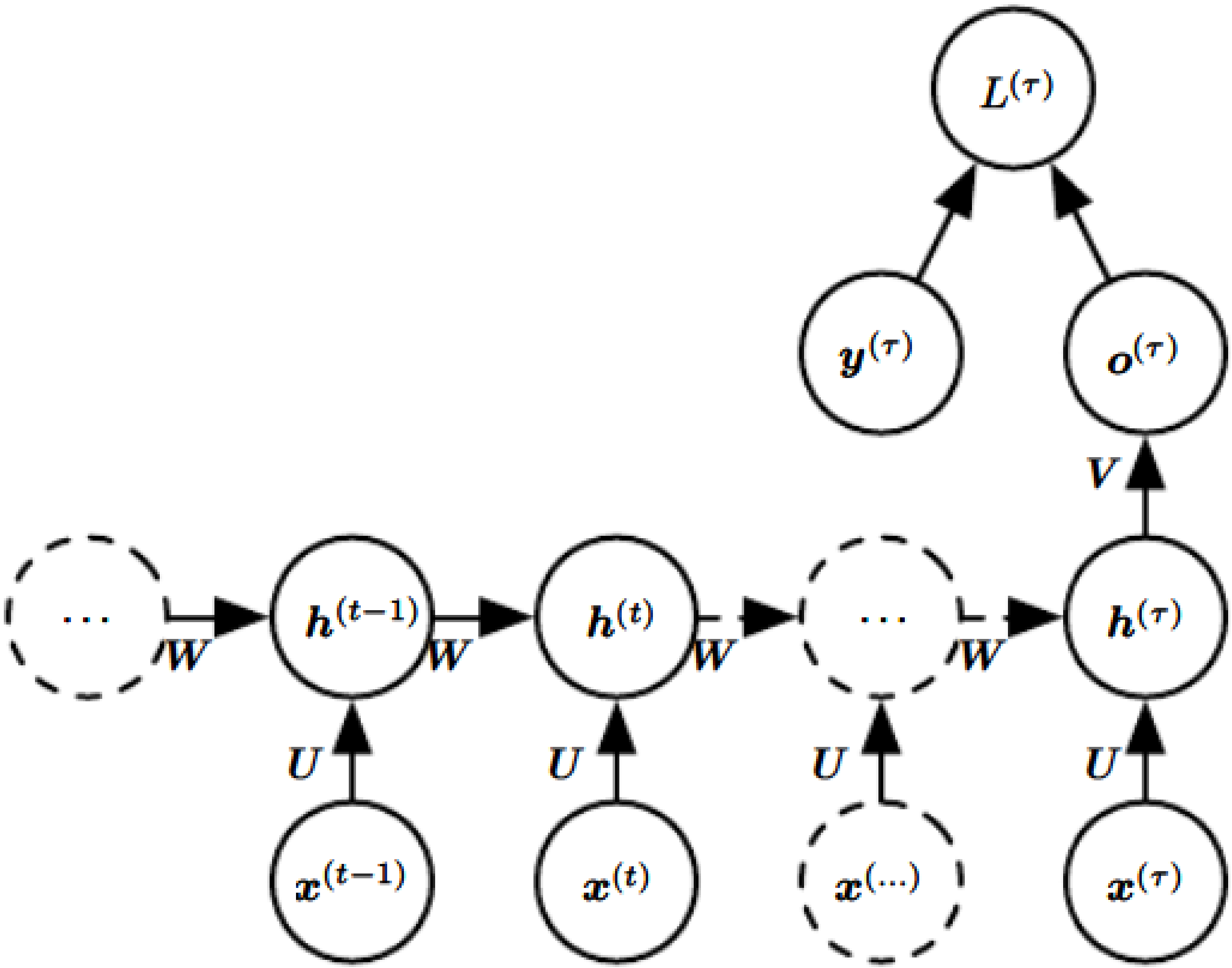
Recurrent networks that produce an output at each time step and have recurrent connections between hidden units

Recurrent Neural Networks



Recurrent networks that produce an output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step

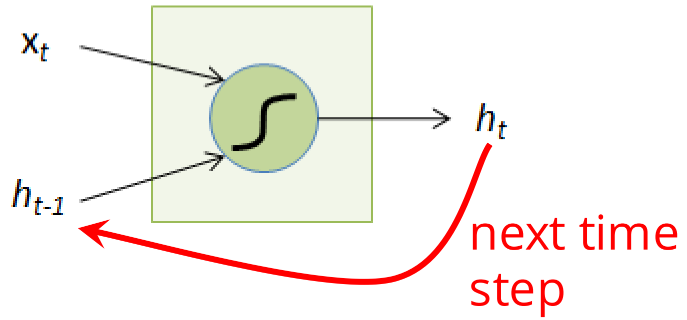
Recurrent Neural Networks



Recurrent networks with recurrent connections between hidden units, that read an entire sequence and then produce a single output

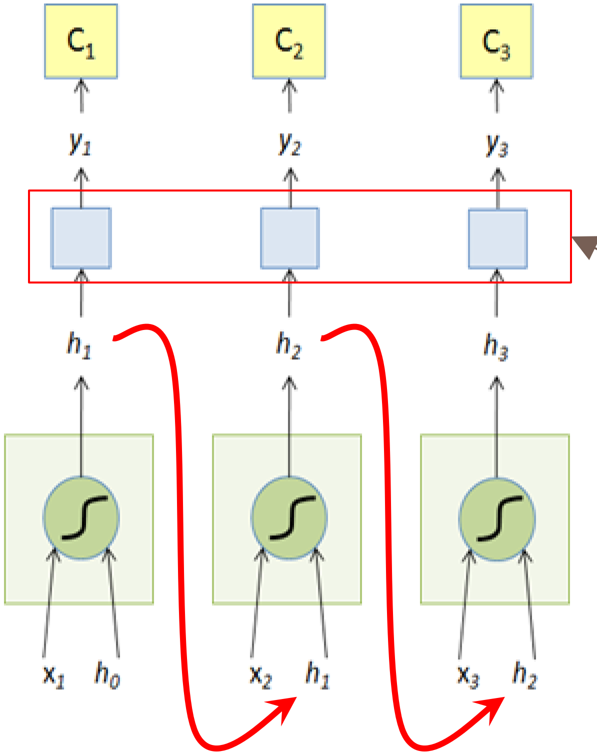
Vanilla RNN cell

x_t: Input at time t
h_{t-1}: State at time t-1



$$h_t = f(W_h h_{t-1} + W_x x_t)$$

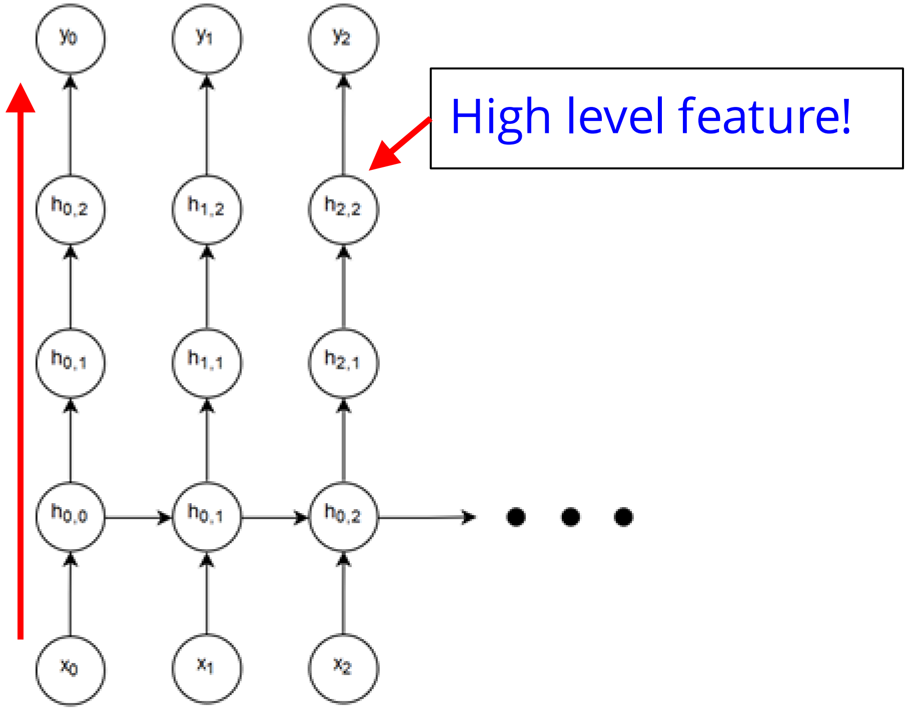
Unfolding



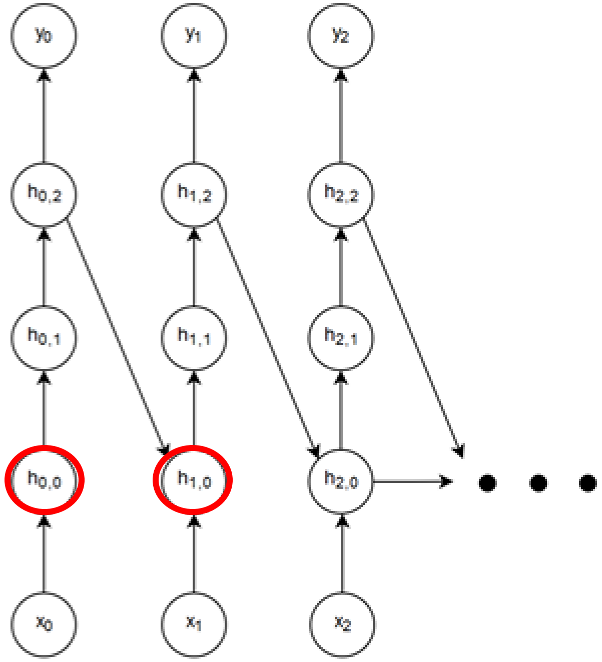
$$h_t = f(W_h h_{t-1} + W_x x_t)$$

Weights shared over time!

Deep RNN

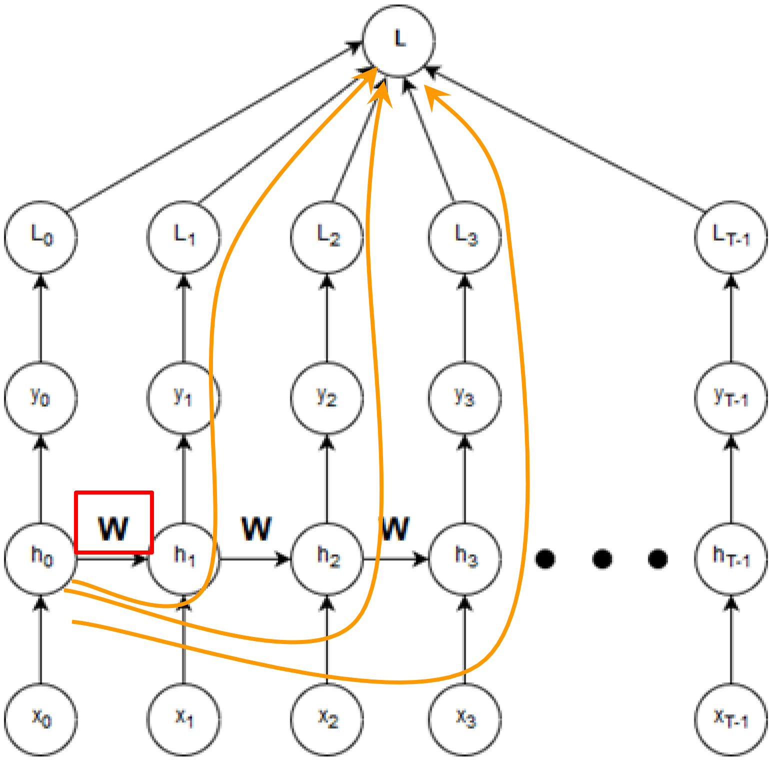


Feedforward depth = 4



Recurrent depth = 3

Backpropagation Trough Time (BPTT)



Objective is to update the weight matrix:

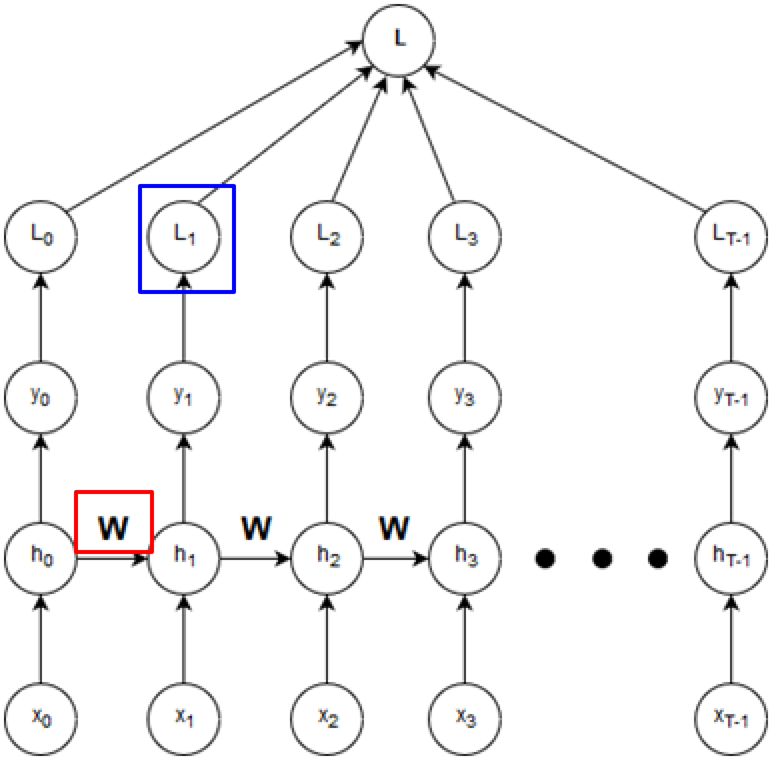
$$\mathbf{W} \rightarrow \mathbf{W} - \alpha \frac{\partial L}{\partial \mathbf{W}}$$

Issue: **W** occurs each timestep
Every path from **W** to L is one dependency

Find all paths from **W** to L!

(note: dropping subscript h from $\mathbf{W}_h$ for brevity)

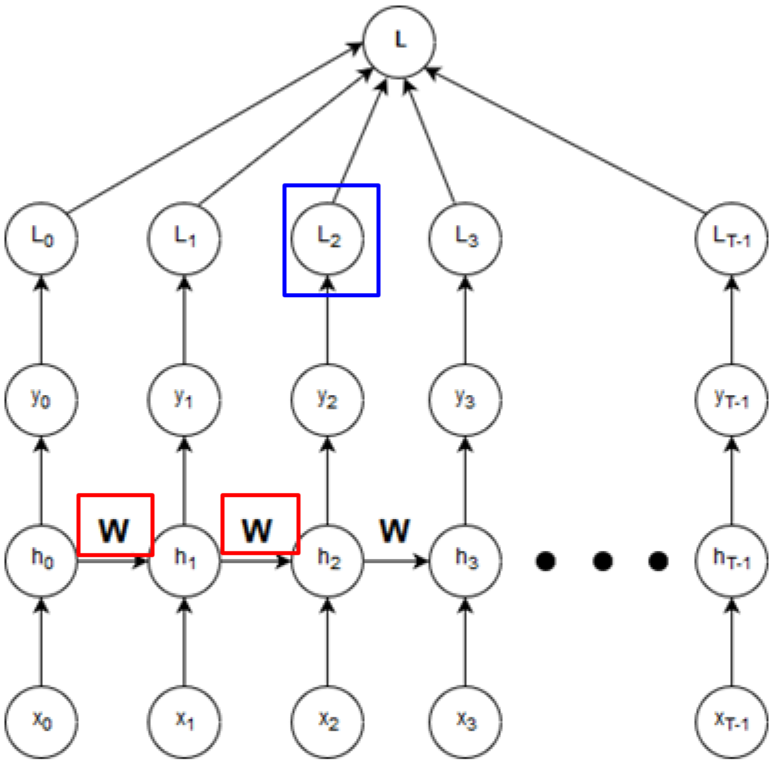
Backpropagation Trough Time (BPTT)



How many paths exist from W to L *through* L_1?

Just 1. Originating at h_0.

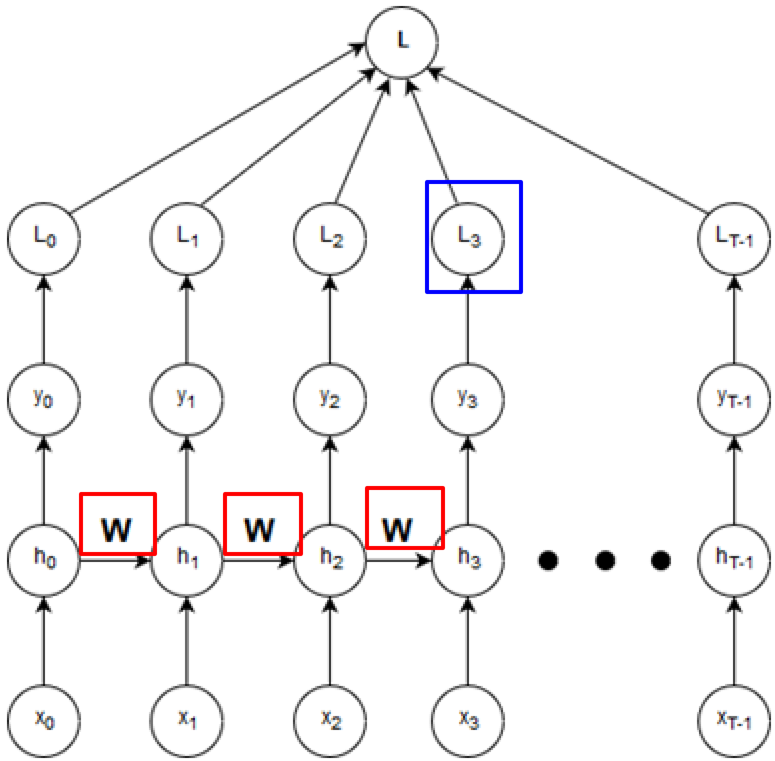
Backpropagation Trough Time (BPTT)



How many paths from **W** to L *through* L_2?

2. Originating at h_0 and h_1.

Backpropagation Trough Time (BPTT)



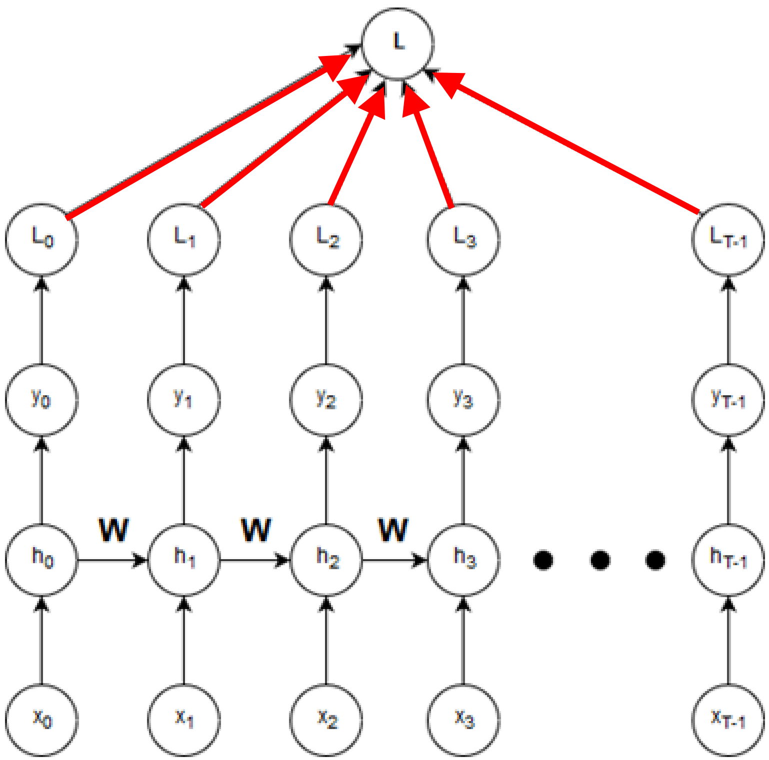
And 3 in this case.

Origin of path = basis for Σ

$$\frac{\partial L}{\partial \mathbf{W}}$$

The gradient has two summations:
1: Over L_j
2: Over h_k

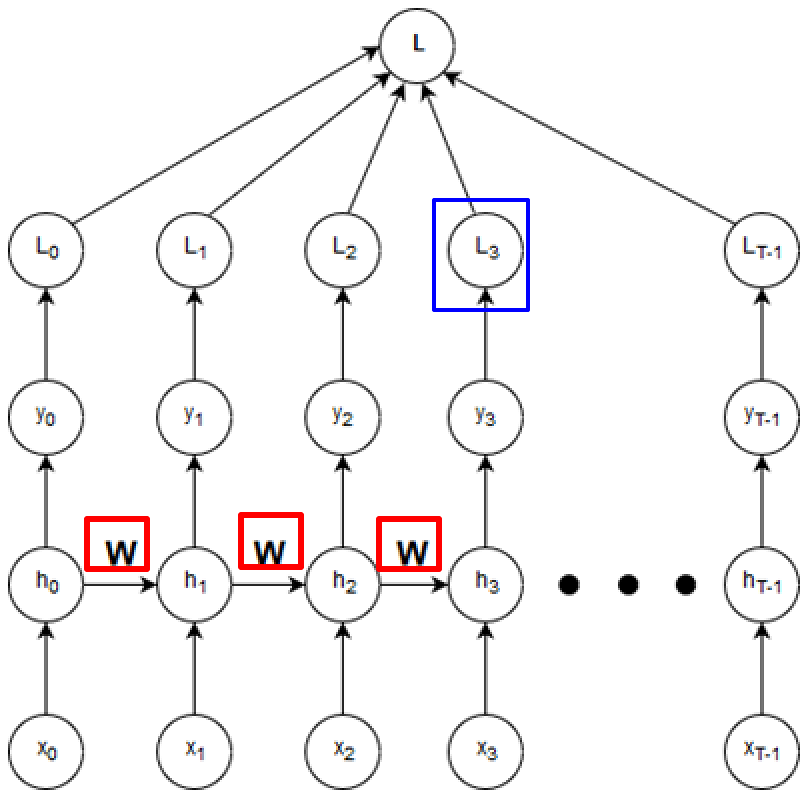
Backpropagation Trough Time (BPTT)



First summation over L

$$\frac{\partial L}{\partial \mathbf{W}} = \sum_{j=0}^{T-1} \frac{\partial L_j}{\partial \mathbf{W}}$$

Backpropagation Trough Time (BPTT)

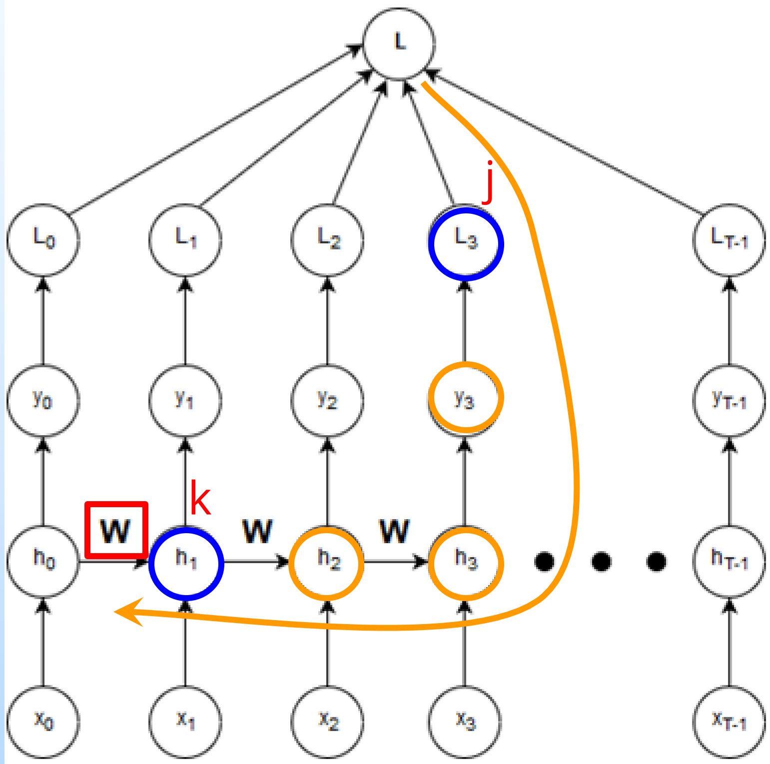


- **Second summation over h:** Each $\mathbf{L}_j$ depends on the weight matrices *before it*

$$\frac{\partial L_j}{\partial \mathbf{W}} = \sum_{k=1}^{j} \boxed{\frac{\partial L_j}{\partial h_k}} \frac{\partial h_k}{\partial \mathbf{W}}$$

L_j depends on all h_k before it.

Backpropagation Trough Time (BPTT)



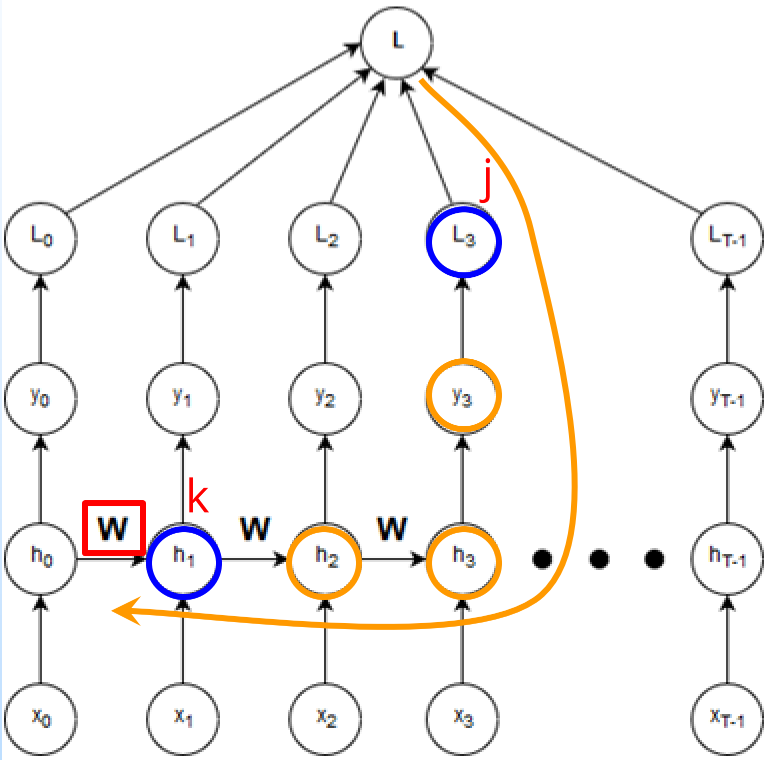
$$\frac{\partial L_j}{\partial W} = \sum_{k=1}^{j} \boxed{\frac{\partial L_j}{\partial h_k}} \frac{\partial h_k}{\partial \mathbf{W}}$$

No explicit of L_j on h_k

Use chain rule to fill missing steps

$$\frac{\partial L_j}{\partial \mathbf{W}} = \sum_{k=1}^{j} \boxed{\frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \frac{\partial h_j}{\partial h_k}} \frac{\partial h_k}{\partial \mathbf{W}}$$

Backpropagation Trough Time (BPTT)



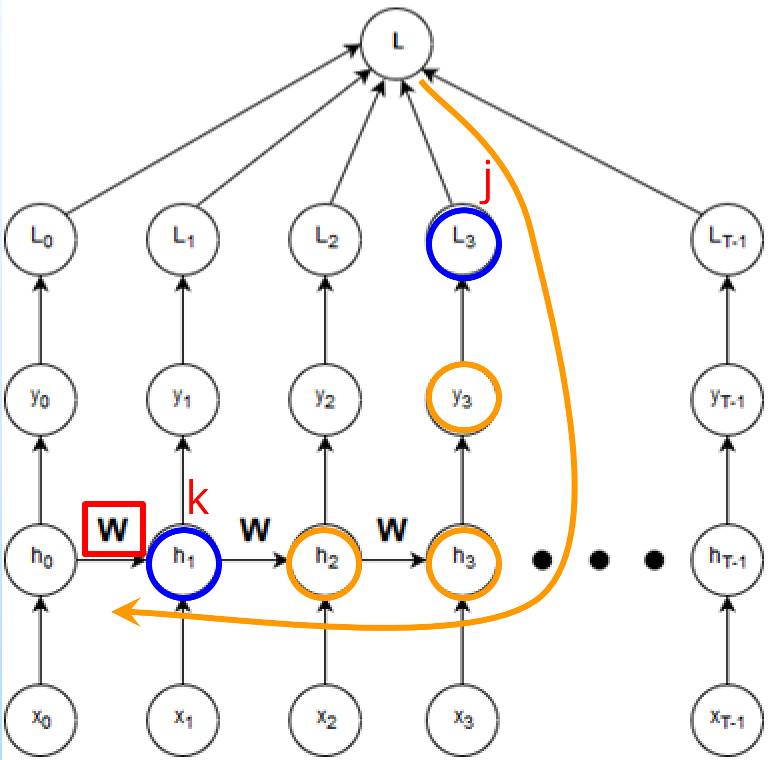
$$\frac{\partial L_j}{\partial W} = \sum_{k=1}^{j} \boxed{\frac{\partial L_j}{\partial h_k}} \frac{\partial h_k}{\partial \mathbf{W}}$$

No explicit of L_j on h_k

Use chain rule to fill missing steps

$$\frac{\partial L_j}{\partial \mathbf{W}} = \sum_{k=1}^{j} \boxed{\frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \frac{\partial h_j}{\partial h_k}} \frac{\partial h_k}{\partial \mathbf{W}}$$

Backpropagation Trough Time (BPTT)



$$\frac{\partial L_j}{\partial W} = \sum_{k=1}^{j} \boxed{\frac{\partial L_j}{\partial h_k}} \frac{\partial h_k}{\partial \mathbf{W}}$$

No explicit of L_j on h_k

Use chain rule to fill missing steps

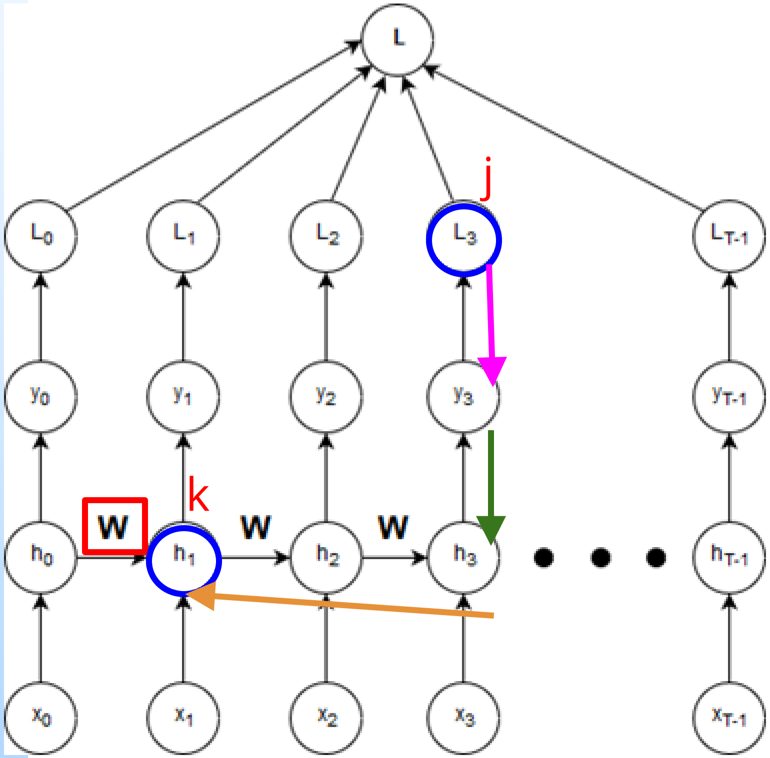
$$\frac{\partial L_j}{\partial \mathbf{W}} = \sum_{k=1}^{j} \boxed{\frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \frac{\partial h_j}{\partial h_k}} \frac{\partial h_k}{\partial \mathbf{W}}$$

Backpropagation Trough Time (BPTT)

$$\frac{\partial L}{\partial \mathbf{W_h}} = \sum_{j=0}^{T-1} \sum_{k=1}^{j} \frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \left(\prod_{m=k+1}^{j} \frac{\partial h_m}{\partial h_{m-1}} \right) \frac{\partial h_k}{\partial \mathbf{W_h}}$$

The final Backpropagation equation

Backpropagation Trough Time (BPTT)



$$\frac{\partial L}{\partial \mathbf{W_h}} = \sum_{j=0}^{T-1} \sum_{k=1}^{j} \boxed{\frac{\partial L_j}{\partial y_j}} \boxed{\frac{\partial y_j}{\partial h_j}} \boxed{\left(\prod_{m=k+1}^{j} \frac{\partial h_m}{\partial h_{m-1}} \right)} \frac{\partial h_k}{\partial \mathbf{W_h}}$$

- Often, to reduce memory requirement, we truncate the network
- Inner summation runs from *j-p* to *j* for some *p* ==> truncated BPTT

Backpropagation Trough Time (BPTT)

$$\frac{\partial L}{\partial W} = \sum_{j=0}^{T-1} \sum_{k=1}^{j} \frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \boxed{\left(\prod_{m=k+1}^{j} \frac{\partial h_m}{\partial h_{m-1}} \right)} \frac{\partial h_k}{\partial \mathbf{W}}$$

$$h_m = f(\mathbf{W}_h h_{m-1} + \mathbf{W}_x x_m)$$

$$\frac{\partial h_m}{\partial h_{m-1}} = \mathbf{W}_h^T \mathrm{diag}\left(f'(\mathbf{W}_h h_{m-1} + \mathbf{W}_x x_m)\right)$$

Expanding the Jacobian

Backpropagation Trough Time (BPTT)

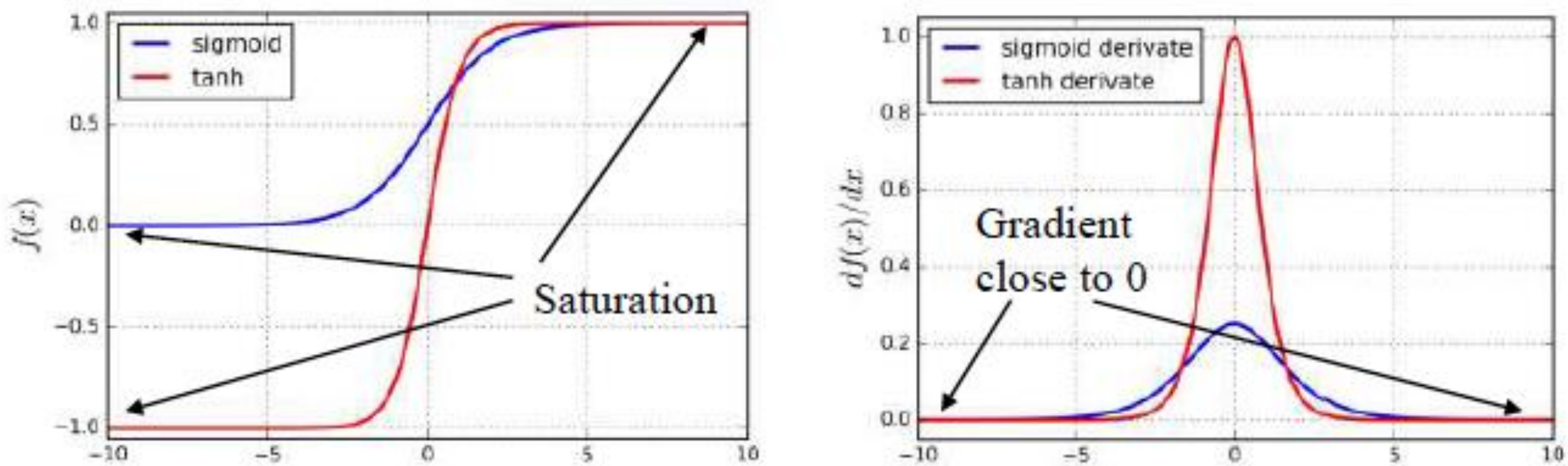
$$\frac{\partial h_j}{\partial h_k} = \prod_{m=k+1}^{j} \boxed{\mathbf{W}_h^T} \boxed{\mathrm{diag}\left(f'(\mathbf{W}_h h_{m-1} + \mathbf{W}_x x_m)\right)}$$

Weight Matrix

Derivative of activation function

Repeated matrix multiplications leads to **vanishing and exploding gradients.**

Vanishing gradients



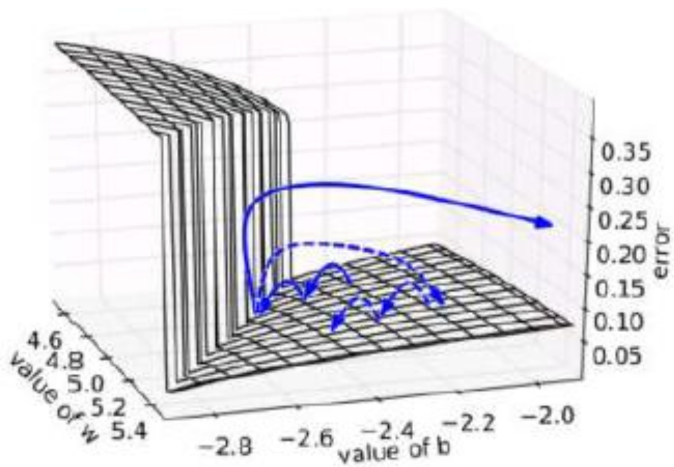
$$\frac{\partial h_t}{\partial h_0} = \frac{\partial h_t}{\partial h_{t-1}} \cdot \ldots \cdot \frac{\partial h_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial h_0}$$

Known problem for deep feed-forward networks.
For recurrent networks (even shallow) makes **impossible to learn long-term** dependencies!

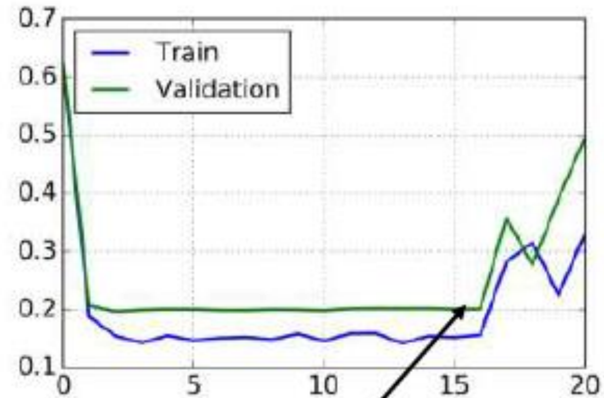
Considerations

- Smaller weight parameters lead to faster gradients vanishing
- Very big initial parameters make the gradient descent to diverge fast (explode)

Exploiding gradients



Pascanou R. et al, On the difficulty of training recurrent neural networks. arXiv (2012)



Network can not converge and weigh parameters do not stabilize

Large increase in the norm of the gradient during training

Diagnostics: NaNs; Cost function large fluctuations

Solutions:
- Use gradient clipping
- Try reduce learning rate
- Change loss function by setting constrains on weights ($L1/L2$ norms)

Eigenvalues and Stability

Consider identity activation function

If Recurrent Matrix $\mathbf{W_h}$ is a diagonalizable:

$$W_h = Q^{-1} * \Lambda * Q$$

Q matrix composed of eigenvectors of W_h

Λ is a diagonal matrix with eigenvalues placed on the diagonals

Computing powers of $\mathbf{W_h}$ is simple:

$$W_h^n = Q^{-1} * \Lambda^n * Q$$

Eigenvalues and Stability

Vanishing gradients

$$\Lambda = \begin{bmatrix} -0.6180 & 0 \\ 0 & 1.6180 \end{bmatrix} \longrightarrow \Lambda^{10} = \begin{bmatrix} \boxed{0.0081} & 0 \\ 0 & \boxed{122.9919} \end{bmatrix}$$

Exploding gradients

$$W_h^n = Q^{-1} * \Lambda^n * Q$$

Fundamental DL problem

- DNNs train difficulties
  - Vanishing gradient
  - Exploiding gradient
- Solutions
  - Previously proposed
  - Unsupervised pre-training
  - Improve network architecture

RNNs - forward propagation

- Assume the hyperbolic tangent activation function
- Initial state $h^{(0)}$
- Update equation

$$\boldsymbol{a}^{(t)} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)}$$

$$\begin{aligned}
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}$$

RNNs – forward propagation

- Total loss

$$
\begin{aligned}
& L\left(\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}\}, \{\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(\tau)}\}\right) \\
= & \sum_t L^{(t)} \\
= & -\sum_t \log p_{\text{model}}\left(y^{(t)} \mid \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}\}\right)
\end{aligned}
$$

Negative log-likelihood

RNNs – Teacher forcing

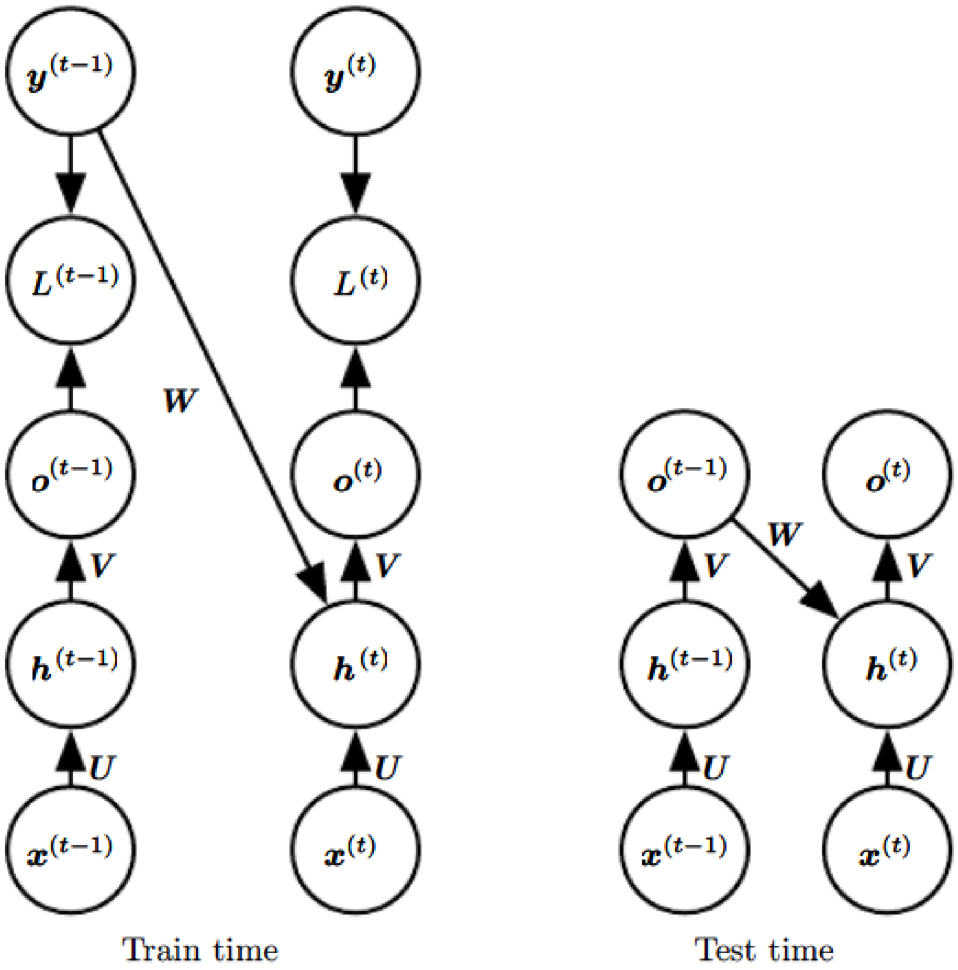


Illustration of teacher forcing

RNNs - learning

- back-propagation through time (BPTT) algorithm
- For each node N we need to compute the gradient recursively
  - based on the gradient computed at nodes that follow it in the graph

$$\nabla_{\mathbf{N}} L$$

  - Start the recursion

$$\frac{\partial L}{\partial L^{(t)}} = 1$$

RNNs - learning

- Gradient on the outputs at time step t, for all i, t,

$$(\nabla_{\boldsymbol{o}^{(t)}} L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}}$$

$$\log \operatorname{softmax}(\boldsymbol{z})_i = z_i - \log \sum_j \exp(z_j)$$

$$\log \sum_j \exp(z_j) \approx \max_j z_j = z_i$$

RNNs - learning

- Backwards starting from the end of the sequence

$$\nabla_{\boldsymbol{h}^{(\tau)}} L = \boldsymbol{V}^\top \nabla_{\boldsymbol{o}^{(\tau)}} L$$

- Back-propagate gradients through time

$$\nabla_{\boldsymbol{h}^{(t)}} L = \left(\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}}\right)^\top (\nabla_{\boldsymbol{h}^{(t+1)}} L) + \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}}\right)^\top (\nabla_{\boldsymbol{o}^{(t)}} L)$$

$$= \boldsymbol{W}^\top (\nabla_{\boldsymbol{h}^{(t+1)}} L) \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t+1)}\right)^2\right) + \boldsymbol{V}^\top (\nabla_{\boldsymbol{o}^{(t)}} L)$$

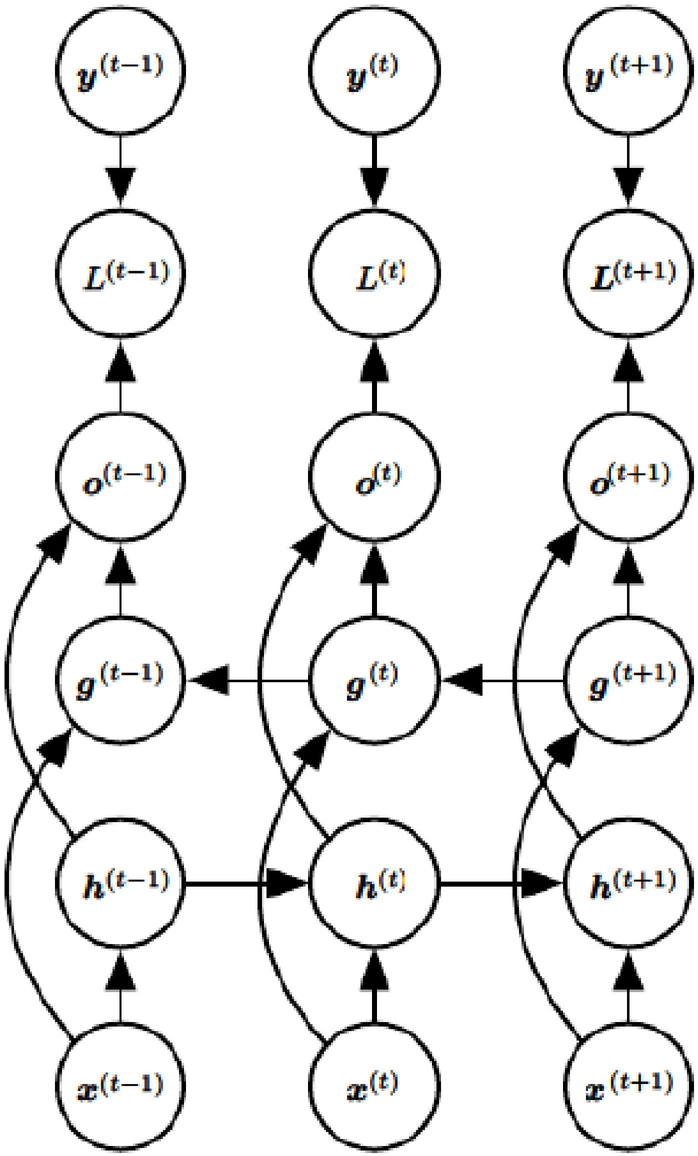
Once the gradients on the internal nodes of the computational graph are obtained, we can obtain the gradients on the parameter nodes

RNNs - learning

- For all the parameters

$$
\begin{aligned}
\nabla_{\boldsymbol{c}} L &= \sum_t \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{c}} \right)^{\top} \nabla_{\boldsymbol{o}^{(t)}} L = \sum_t \nabla_{\boldsymbol{o}^{(t)}} L \\
\nabla_{\boldsymbol{b}} L &= \sum_t \left(\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{b}^{(t)}} \right)^{\top} \nabla_{\boldsymbol{h}^{(t)}} L = \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \nabla_{\boldsymbol{h}^{(t)}} L \\
\nabla_{\boldsymbol{V}} L &= \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^{(t)}} \right) \nabla_{\boldsymbol{V}} o_i^{(t)} = \sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \boldsymbol{h}^{(t)\top} \\
\nabla_{\boldsymbol{W}} L &= \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{\boldsymbol{W}^{(t)}} h_i^{(t)} \\
&= \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \boldsymbol{h}^{(t-1)\top} \\
\nabla_{\boldsymbol{U}} L &= \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{\boldsymbol{U}^{(t)}} h_i^{(t)} \\
&= \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \boldsymbol{x}^{(t)\top}
\end{aligned}
$$

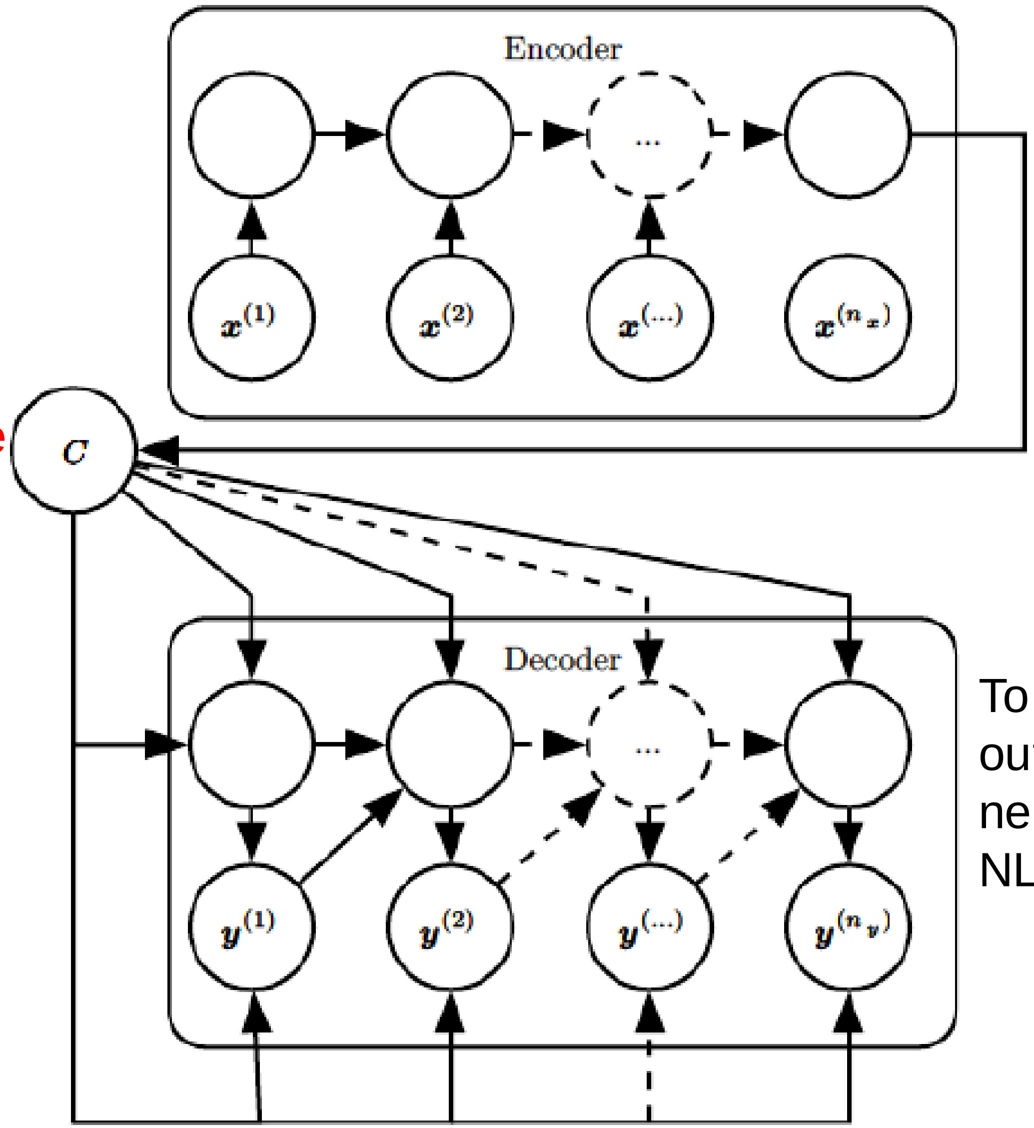
RNNs - Bidirectional



prediction of y(t) which may depend on the whole input sequence e.g., speech recognition

RNNs – Encoder and Decoder

Context vector or sequence

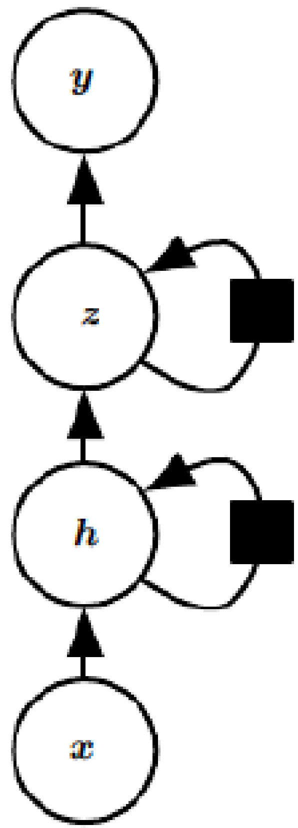


encoder-decoder or sequence-to-sequence RNN architecture

To map input sequence to an output sequence which is not necessarily the same lenght: NLP, speech recongition, …

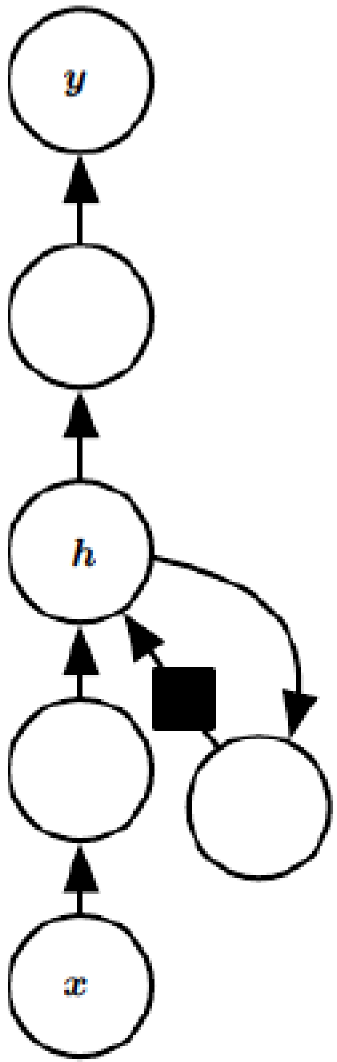
Attention meccanism to C could be added

Deep Recurrent Networks



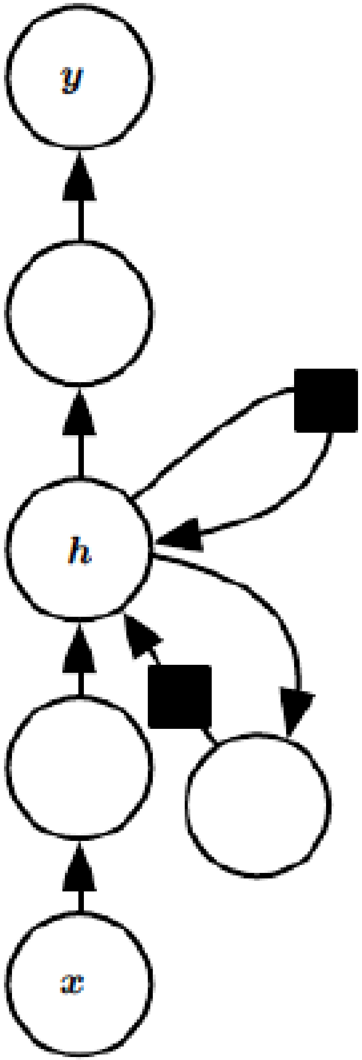
The hidden recurrent state can be broken down into groups organized hierarchically

Deep Recurrent Networks



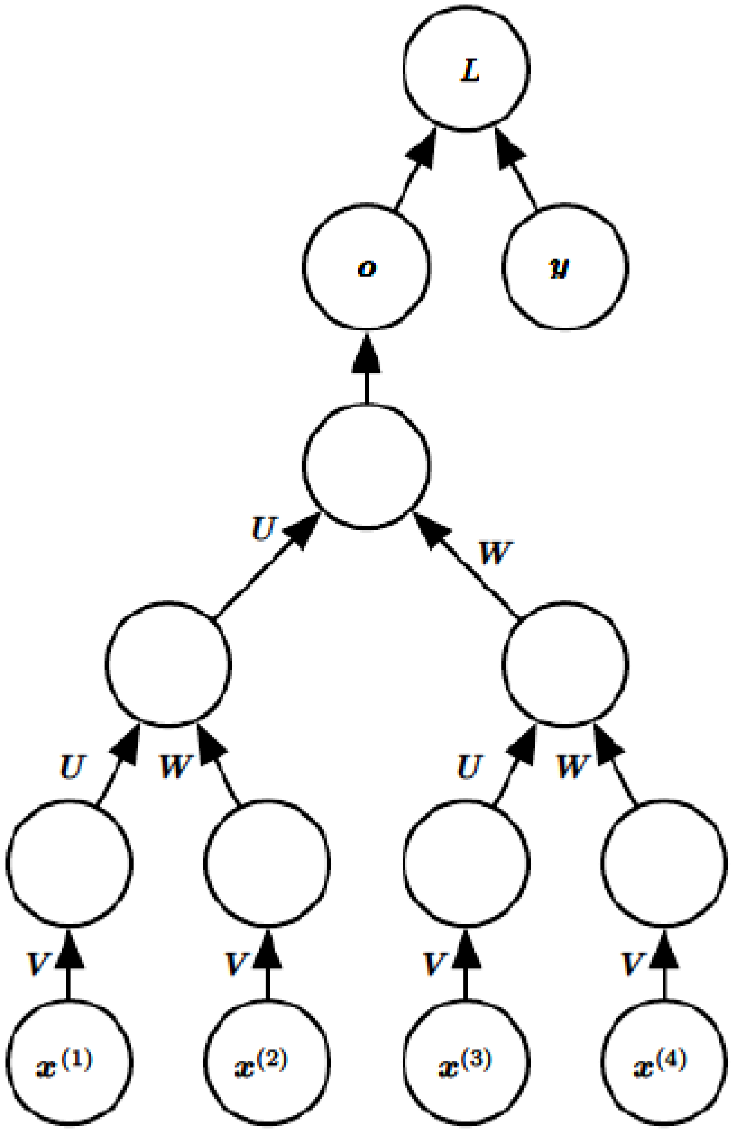
Deeper computation (e.g., an MLP) can be introduced in the input-to-hidden, hidden-to-hidden and hidden-to-output parts. This may lengthen the shortest path linking different time steps.

Deep Recurrent Networks



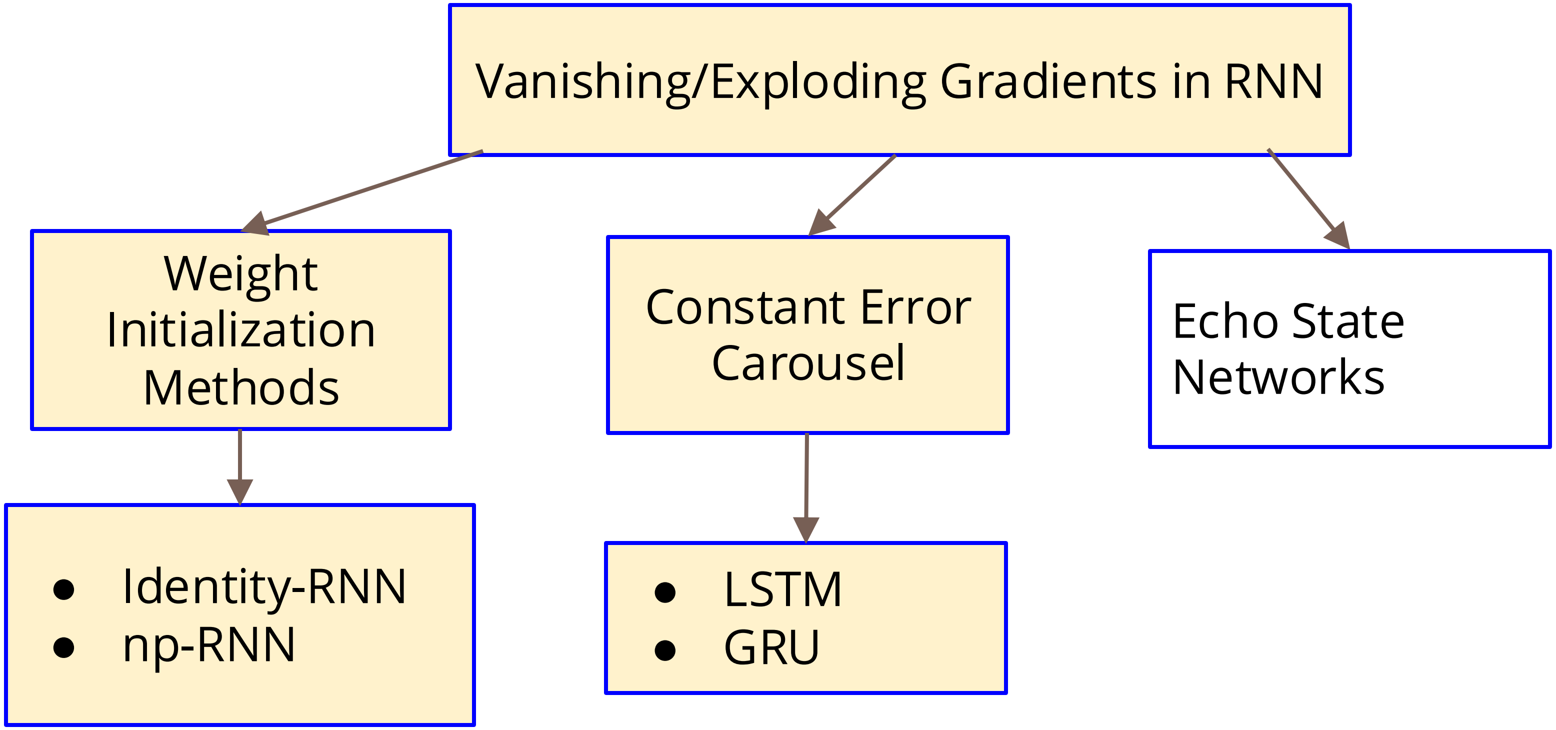
The path-lengthening effect can be mitigated by introducing skip connections

Recursive NNs



Generalization of recurrent networks
Applied for structurated data

Long-Term dependendencies



Long-Term dependendencies

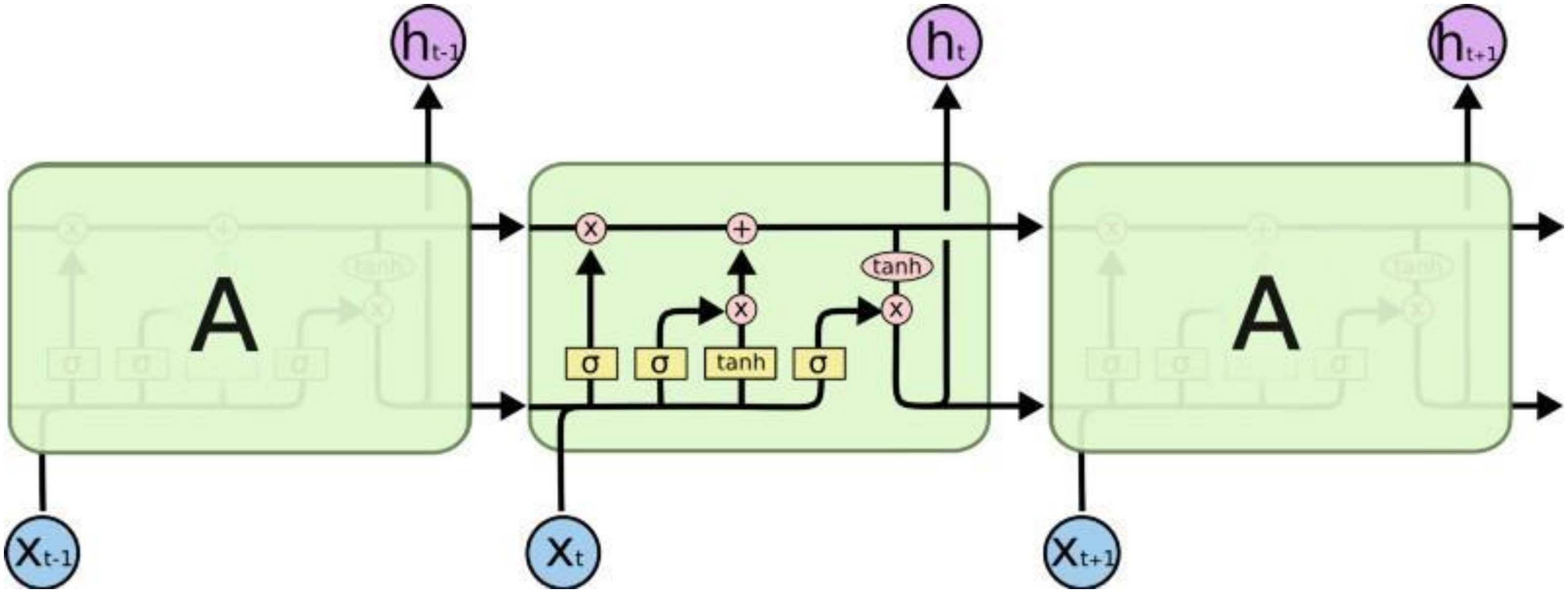
- Random W_h initialization of RNN has no constraint on eigenvalues
  - vanishing or exploding gradients in the initial epoch
- Careful initialization of W_h with suitable eigenvalues
  - allows the RNN to learn in the initial epochs
  - hence can generalize well for further iterations

Long-Term dependendencies

- Trick #1(IRNN)
  - W_h initialized to Identity
  - Activation function: ReLU

- Trick# 2 (np-RNN)
  - Wh positive definite (+ve real eigenvalues)
  - At least one eigenvalue is 1, others all less than equal to one
  - Activation function: ReLU

Long Short-Term Memory



Gated RNNs

- Gated RNNs
  - Long Short-Term memory
  - Gated Recurrent Unit

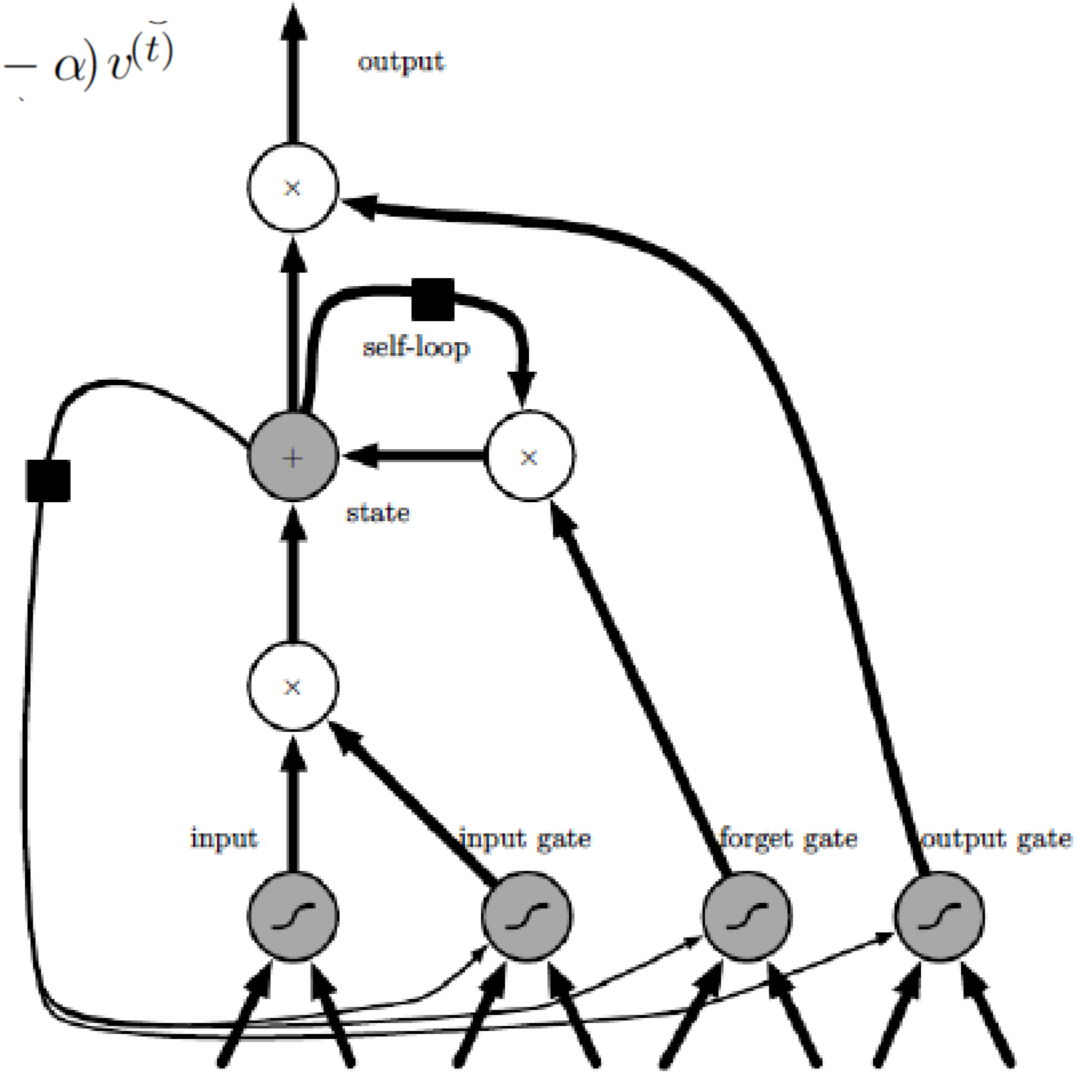
- Idea
  - creating paths through time that have derivatives that neither vanish nor explode

Gated RNNs

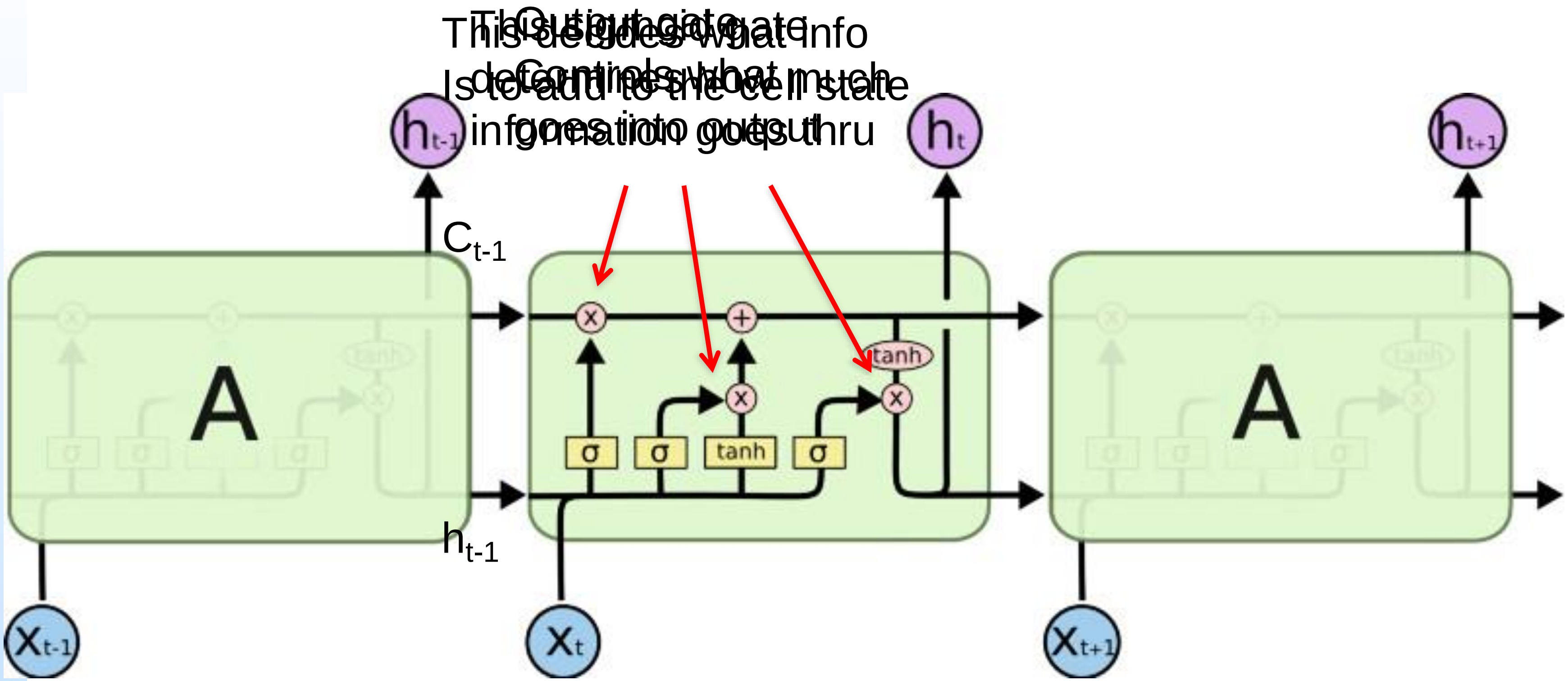
Leaky Units

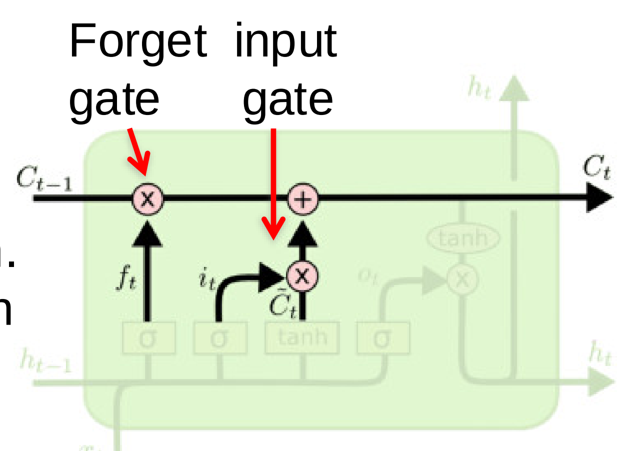
$$\mu^{(t)} \leftarrow \alpha \mu^{(t-1)} + (1-\alpha) v^{(t)}$$

linear self-connections and a weight near one on these connections



LSTM cell



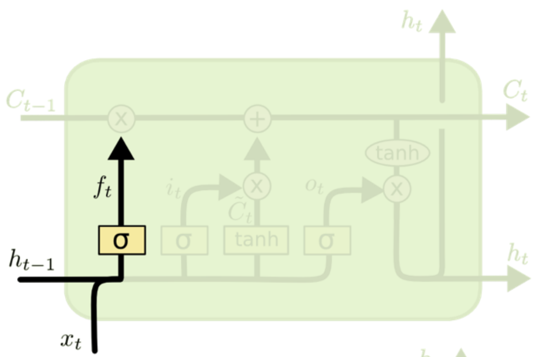


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

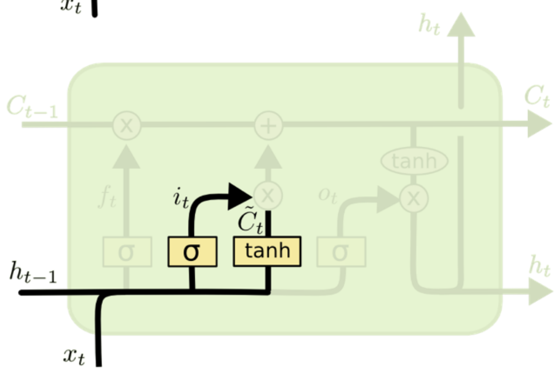
The core idea is this cell state C_t, it is changed slowly, with only minor linear interactions. It is very easy for information to flow along it unchanged.

Why sigmoid or tanh:
Sigmoid: 0,1 gating as switch.
Vanishing gradient problem in LSTM is handled already.

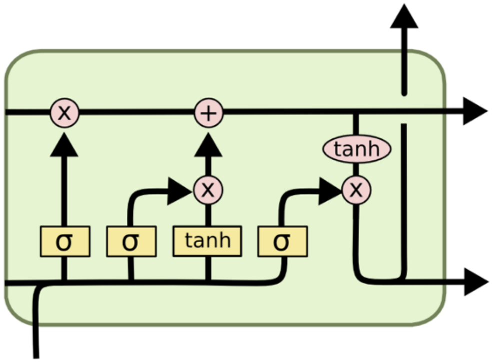
LSTM cell



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

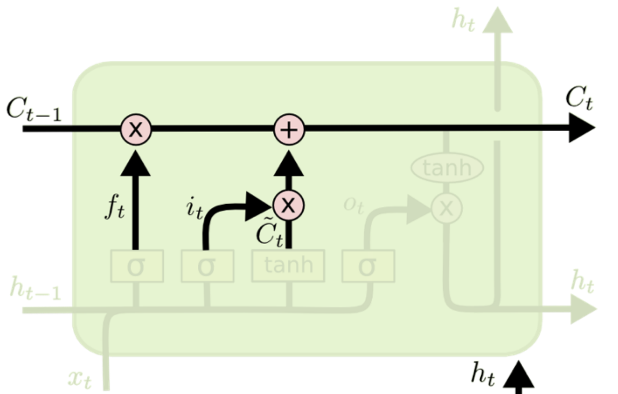


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$



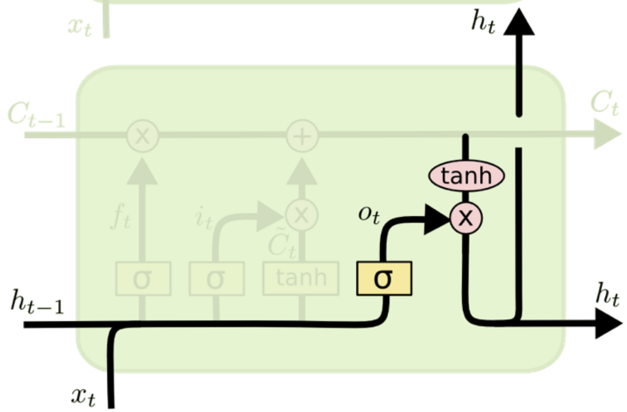
i_t decides what component is to be updated.
C'$_t$ provides change contents

LSTM cell



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

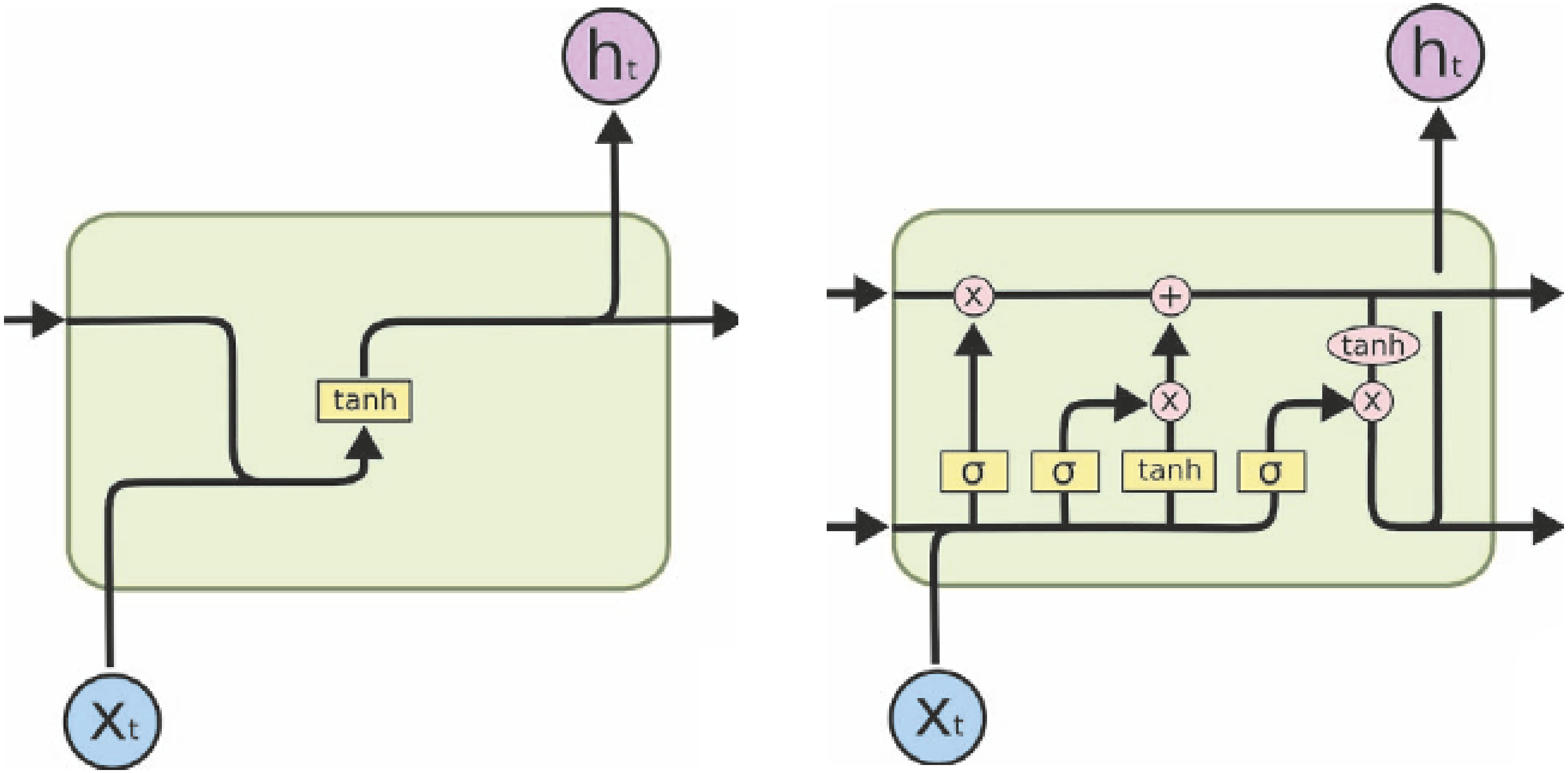
Updating the cell state



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Decide what part of the cell state to output

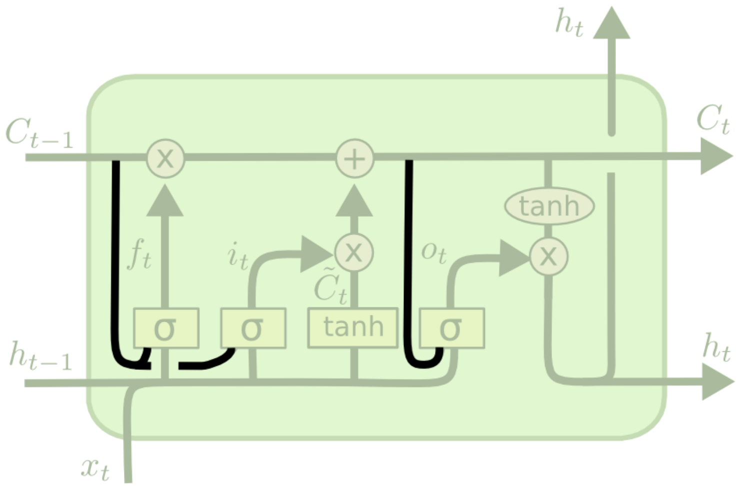
RNN vs LSTM



(a) RNN

(b) LSTM

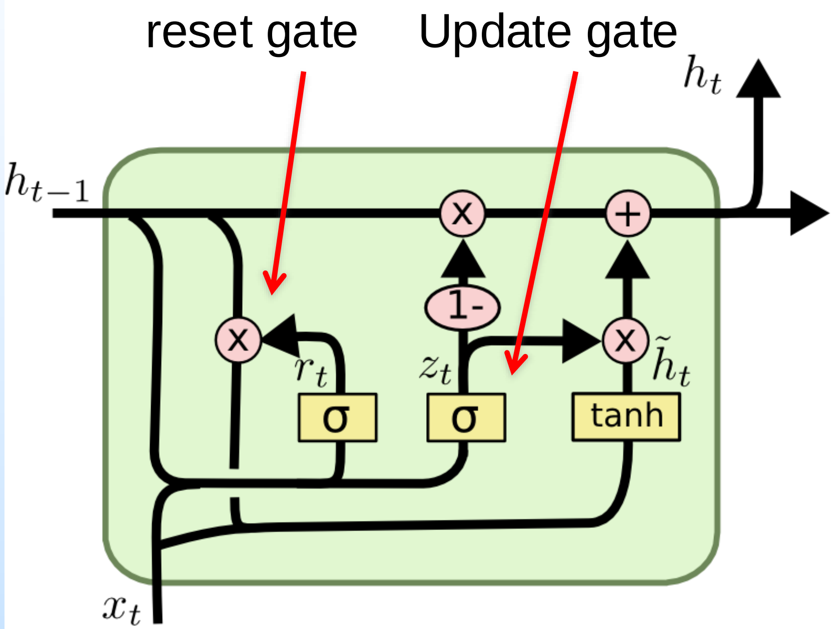
Peephole LSTM



$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

Allows "peeping into the memory". Can learn the fine distinction between sequences of spikes separated by either 50 or 49 discrete time steps

Gated Recurrent Unit (GRU)



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

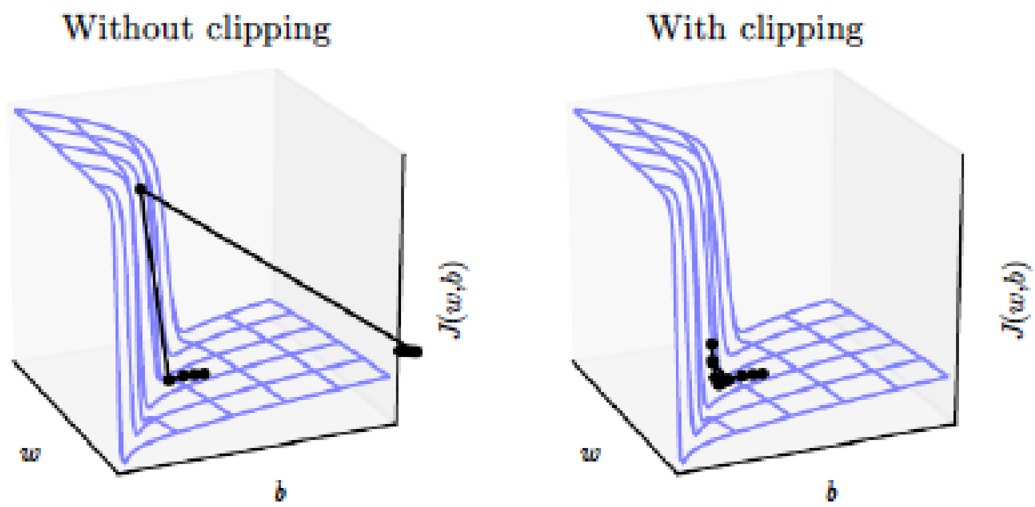
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

It combines the forget and input into a single update gate.
It also merges the cell state and hidden state. This is simpler than LSTM. There are many other variants too.

Clipping gradients



parameter gradient is very large

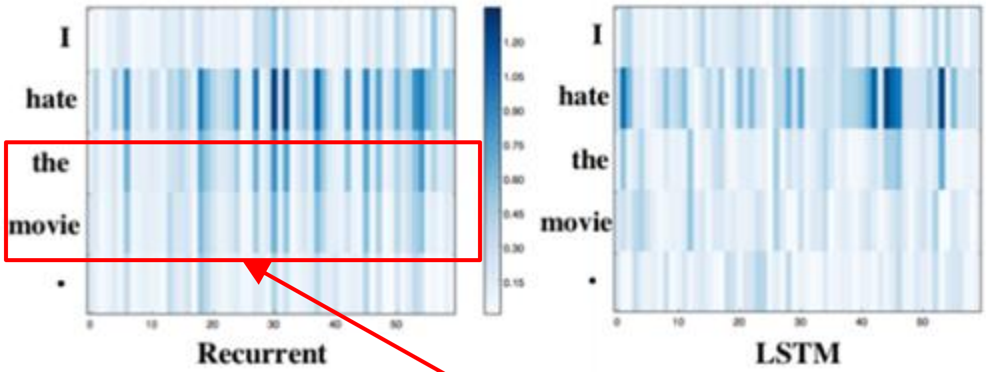
"landscape" in which one finds "cliffs"

$$\text{if } ||\boldsymbol{g}|| > v$$
$$\boldsymbol{g} \leftarrow \frac{\boldsymbol{g}v}{||\boldsymbol{g}||}$$

Clipping the gradient

RNN vs LSTM

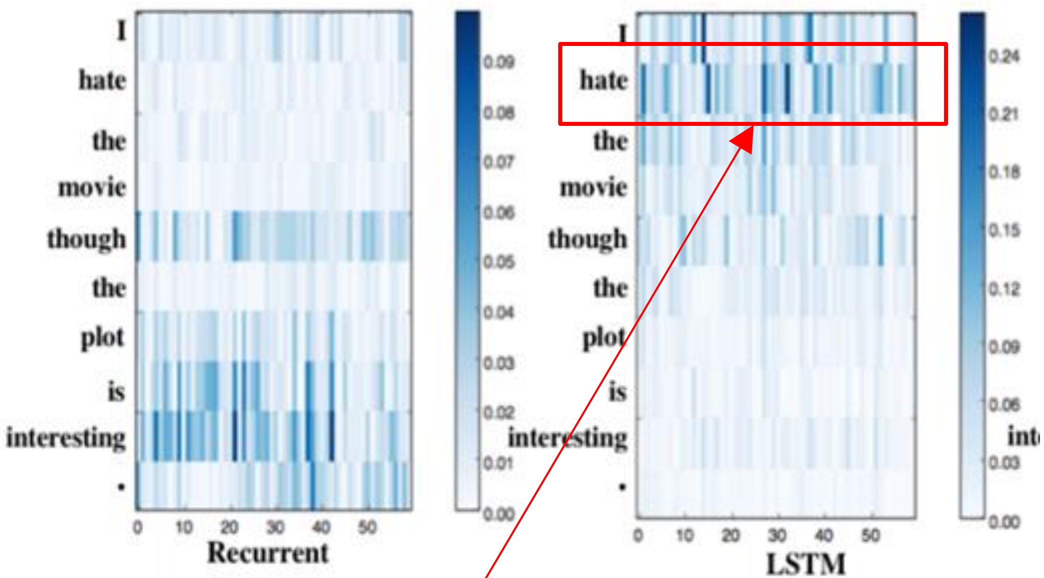
Saliency Heatmap



Recent words more salient

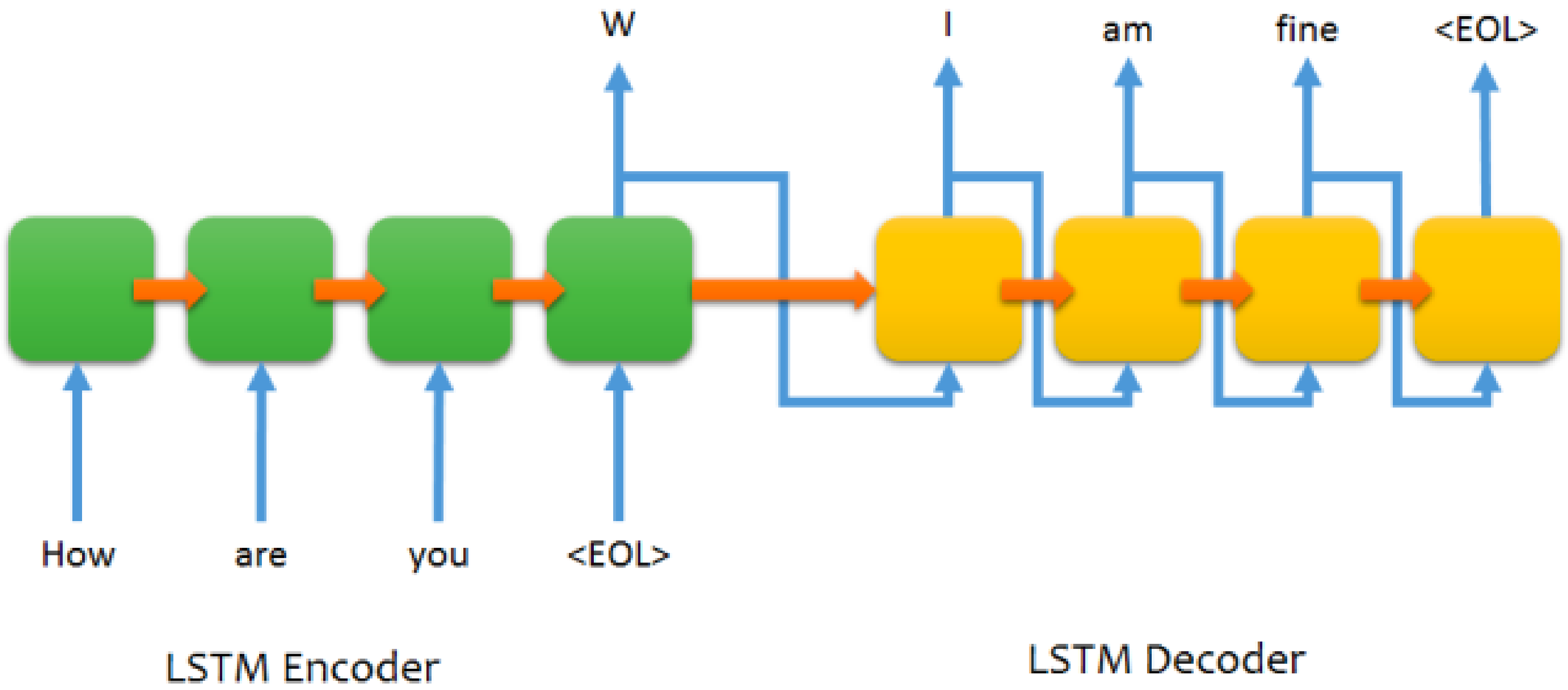
RNN vs LSTM

Saliency Heatmap



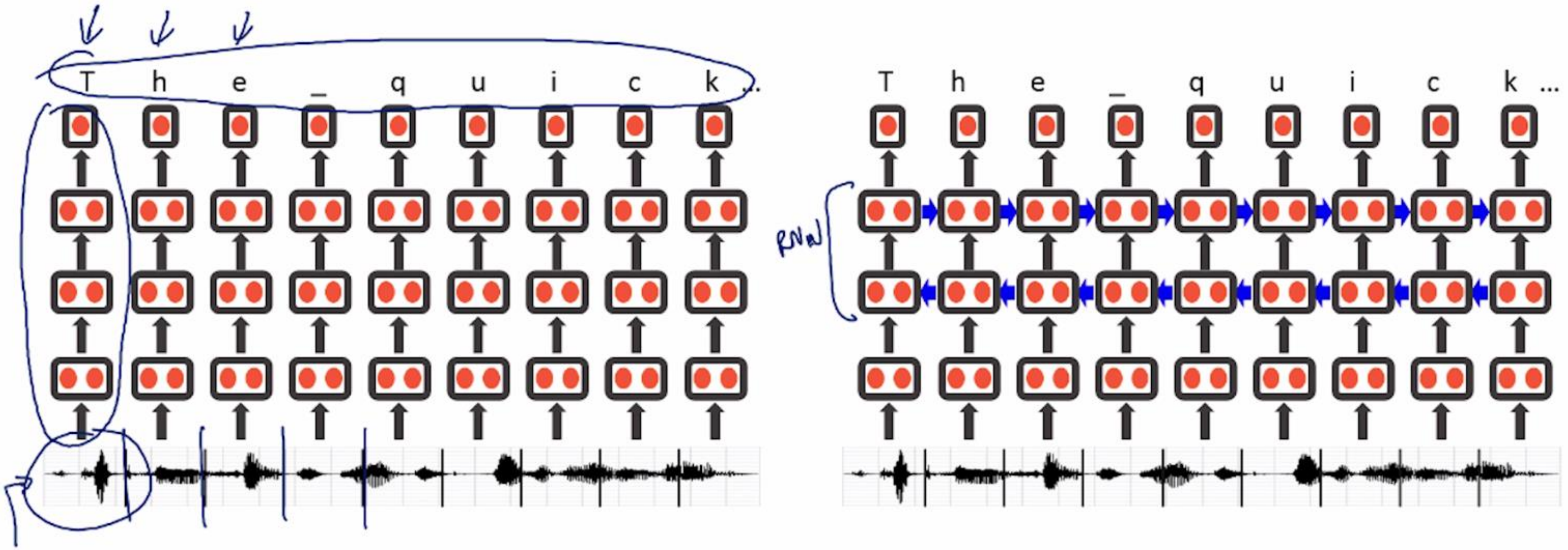
LSTM captures long term dependencies

Sequence to sequence chat model



Speech recognition RNN

Speech recognition example (Deep Speech)

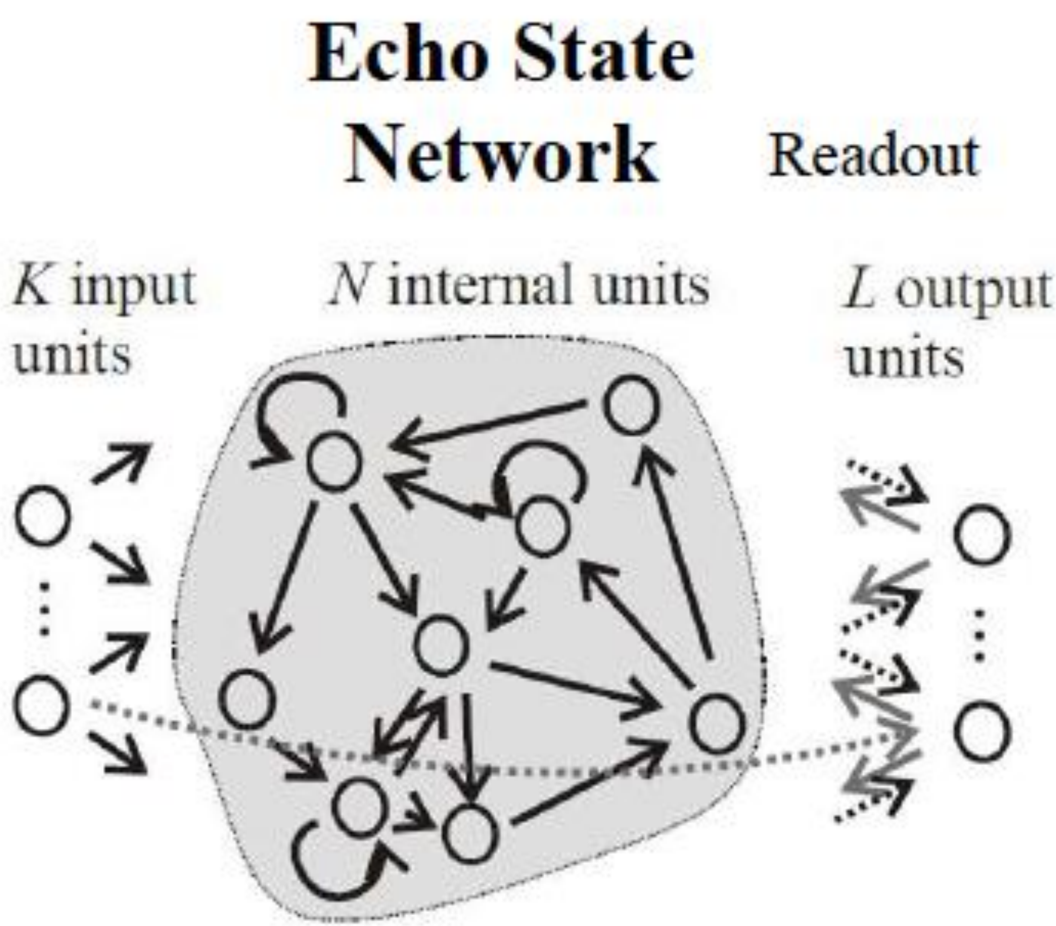


Reservoir computing

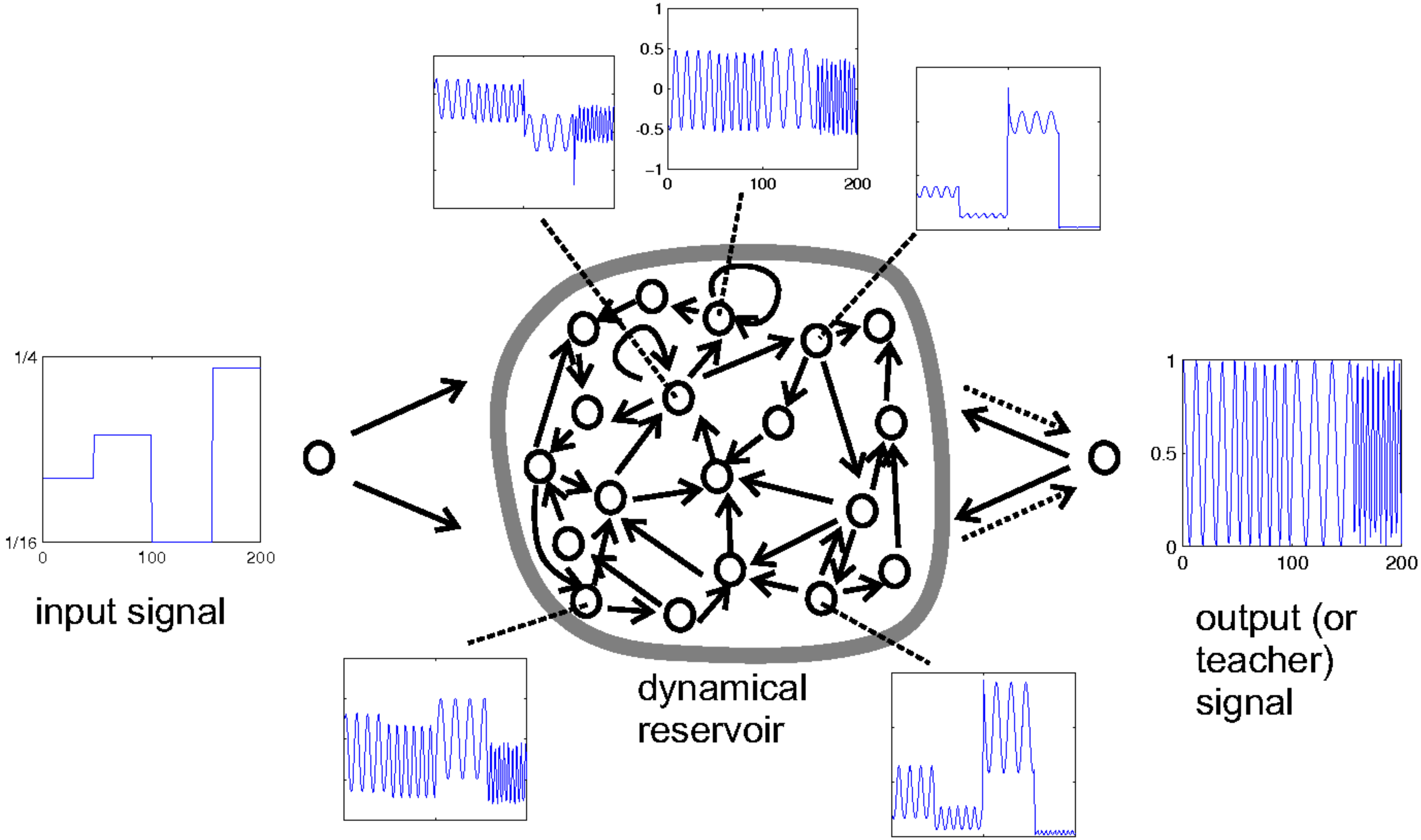
- The equivalent idea for RNNs
  - fix the input- hidden connections and the hidden-hidden connections at random values
  - only learn the hidden-output connections
- The learning is then very simple (assuming linear output units)
- Its important to set the random connections very carefully so the RNN does not explode or die
- See also Liquid State Machine

Reservoir computing

Herbert Jaeger, 2001



Reservoir computing



ML – Recurrent NNs