



COMPUTATIONAL INTELLIGENCE

DEEP LEARNING

**Recurrent Neural Networks and Long Short-Term Memory
for Learning Sequences and Natural Language Processing**



Adrian Horzyk
horzyk@agh.edu.pl

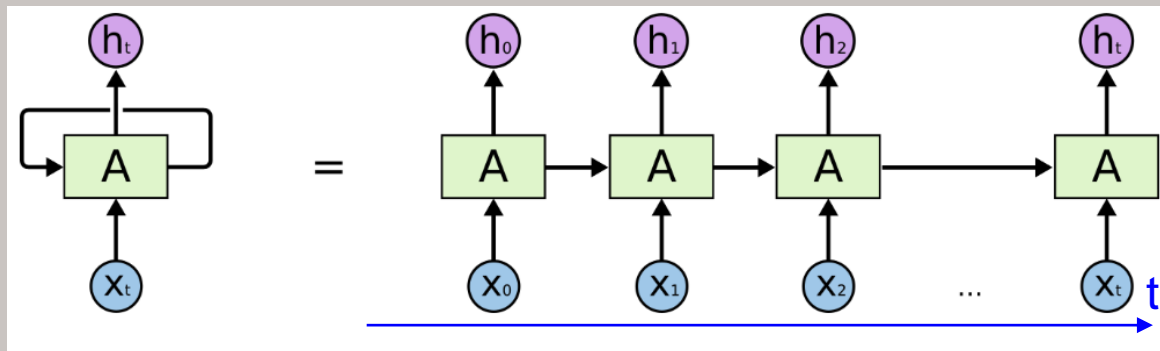


**AGH University of
Science and Technology
Krakow, Poland**

Introduction



- **Human thinking process** does not start from scratch every second for each pattern as is usually processed in CNNs and classic artificial neural networks – what is their major shortcoming between others.
- We always take into account **previous words, situations, and states** of our brains, not throwing away all previous thoughts during e.g. speech recognition, machine translation, entity names recognition, sentiment classification, music generation, or image captioning.
- Our intelligence works so well because it is not started again and again for every new situation but incorporates the **knowledge** that is gradually formed **in time**. Thanks to it, all next intelligent processes take into account our **previous experiences**.
- **Recurrent neural networks** address this issue, implementing various loops, allowing information to persist, and gradually processing data in time (following time steps).



- We can take into account previous state of the network, previous inputs and/or previous outputs during computations.
- This **chain-like nature** reveals that recurrent neural networks are intimately related to **sequences** and **lists**, and are the natural neural network architecture for such data.

Introduction to NLP and Words Representation



- **Natural Language Processing (NLP)** includes various tasks of language analysis, understanding, translation, generation, classification and clustering, so we need to operate on words.
- We usually use any kind of a word dictionary (a vocabulary of the processed language) and each word from the given vocabulary can be represented as a **one-hot vector**, which is the vector consisting of zeros except to the single position representing a given word equal to 1:

| No | Dictionary | a | and | word | zyzzyva |
|--------|------------|-----|-----|------|---------|
| 1 | a | 1 | 0 | 0 | 0 |
| | ... | ... | ... | ... | ... |
| 982 | and | 0 | 1 | 0 | 0 |
| | ... | ... | ... | ... | ... |
| 132847 | word | 0 | 0 | 1 | 0 |
| | ... | ... | ... | ... | ... |
| 171476 | zyzzyva | 0 | 0 | 0 | 1 |

- The one-hot vectors are often used to represent a sequence of words on the inputs of the recurrent neural networks as well as other types of neural networks.

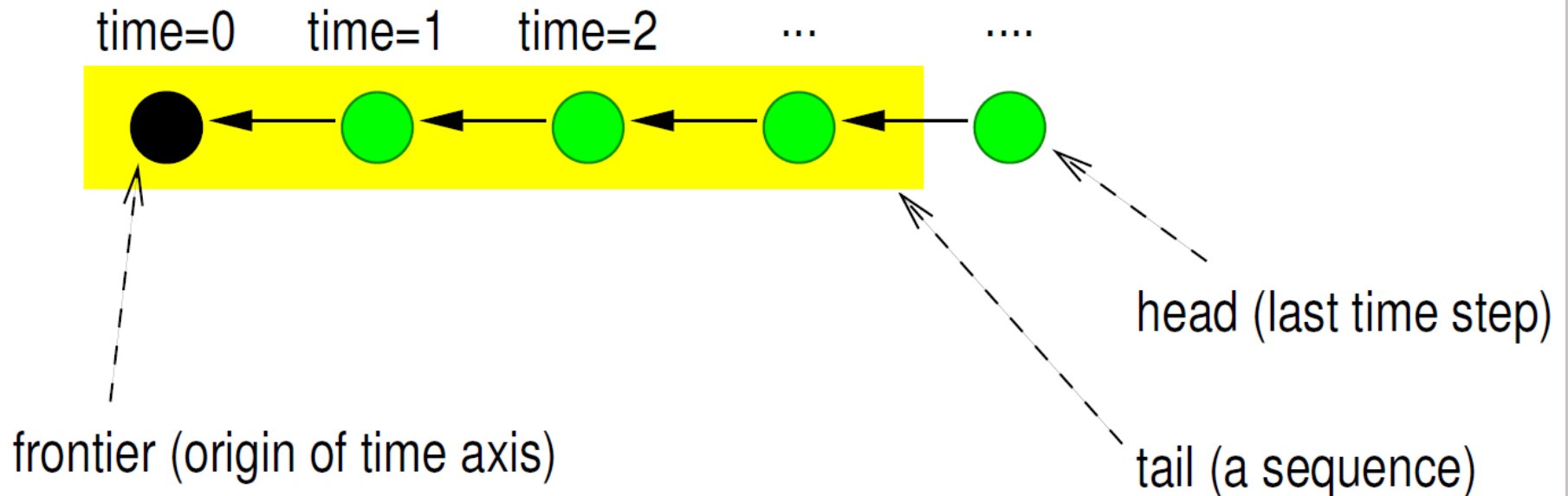
Sequential Data and Domains



Sequential patterns differ from static patterns because:

- successive data (points) are strongly correlated,
- the succession of data is crucial from their recognition/classification point of view.

A sequence can be defined using mathematical induction as an external vertex or an ordered pair (t,h) where the head h is a vertex and the tail t is a sequence:

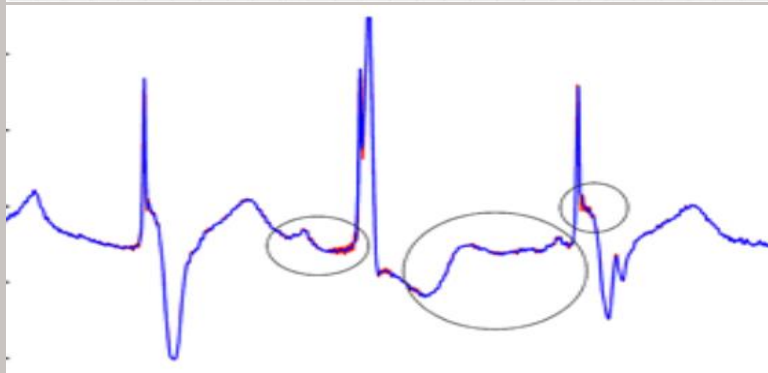
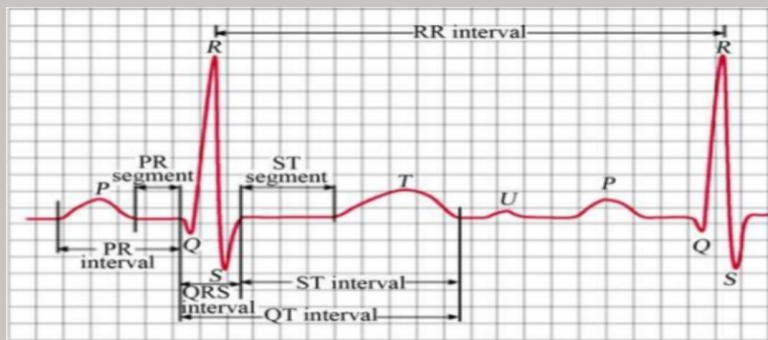


Examples of Sequential Data

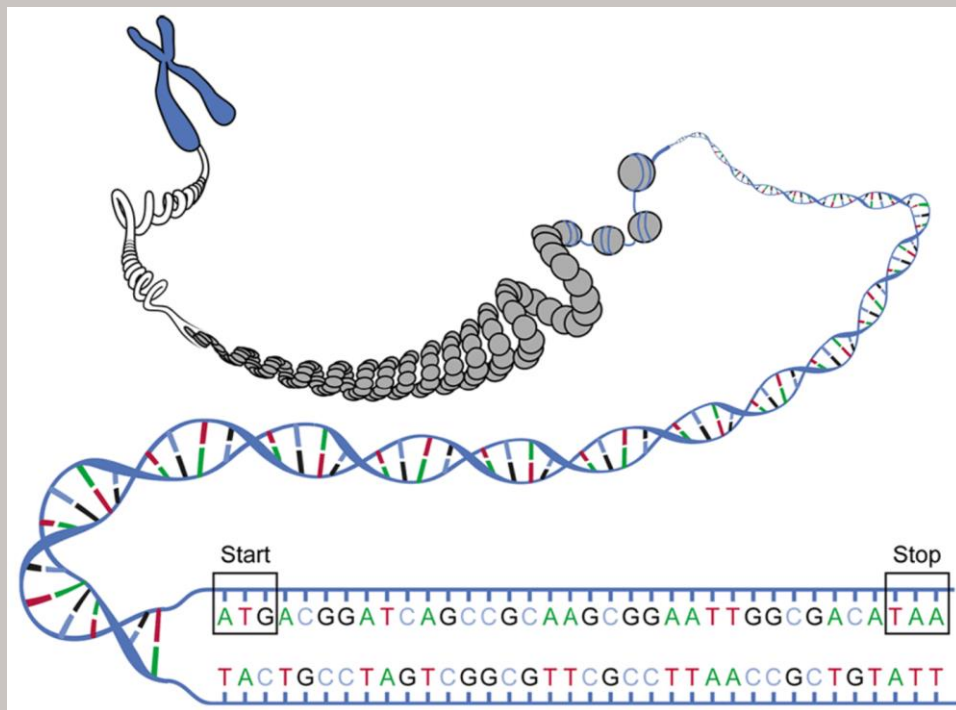


Examples of sequential data where context is defined by sequences of data:

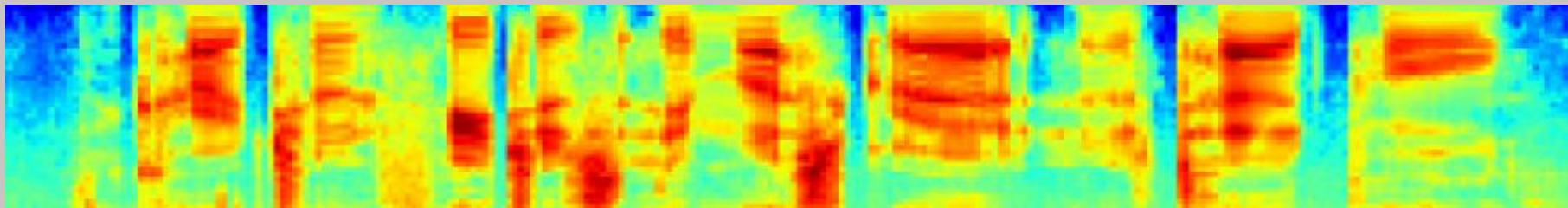
ECG signals:



Genes and Chromosomes:



Speech signals (sequences of letters, words, phonemes, or audio time data):

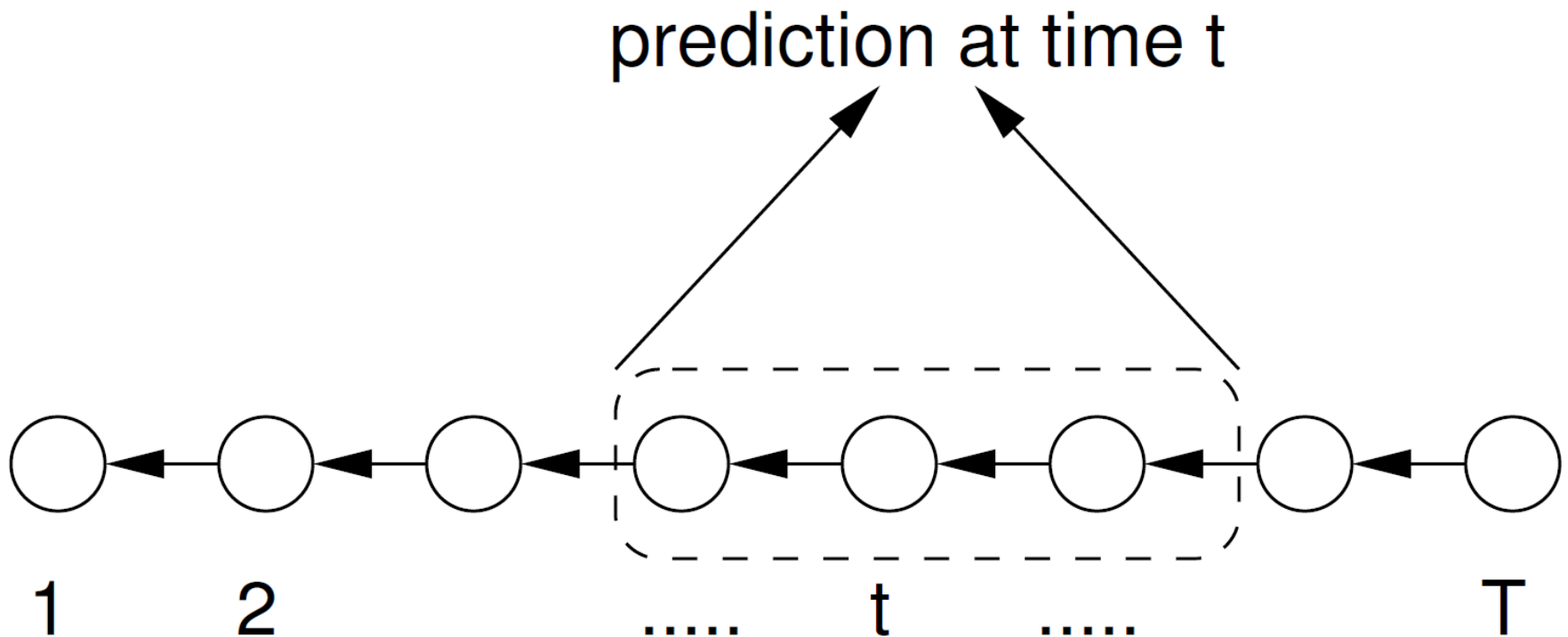


LEARNING SEQUENCES



Sequences usually **model processes in time (actions, movements)** and are sequentially processed in time to predict next data (conclusions, reactions).

Sequences can have **variable length** but typical machine learning models use a fixed number of inputs (fixed-size window) as a prediction context:

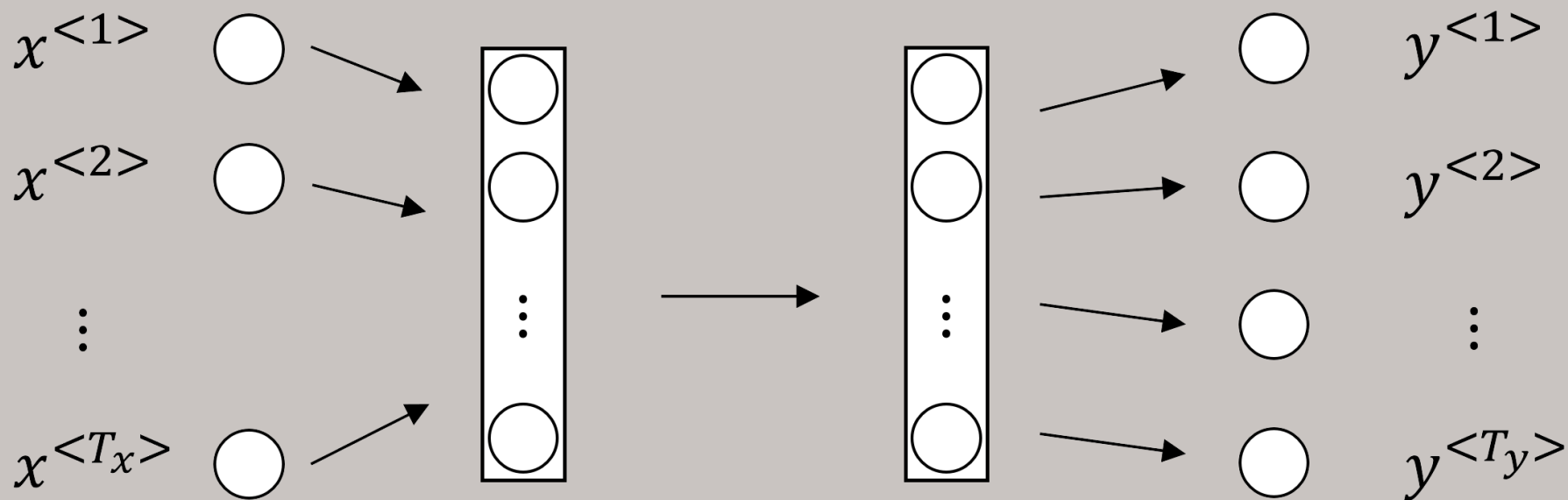


A standard network will not work!



When dealing with sequential data (like sentences of words):

- Inputs and outputs can usually be different lengths in different examples.
- The same words in the different examples do not share the same inputs and features learned across different positions of text.



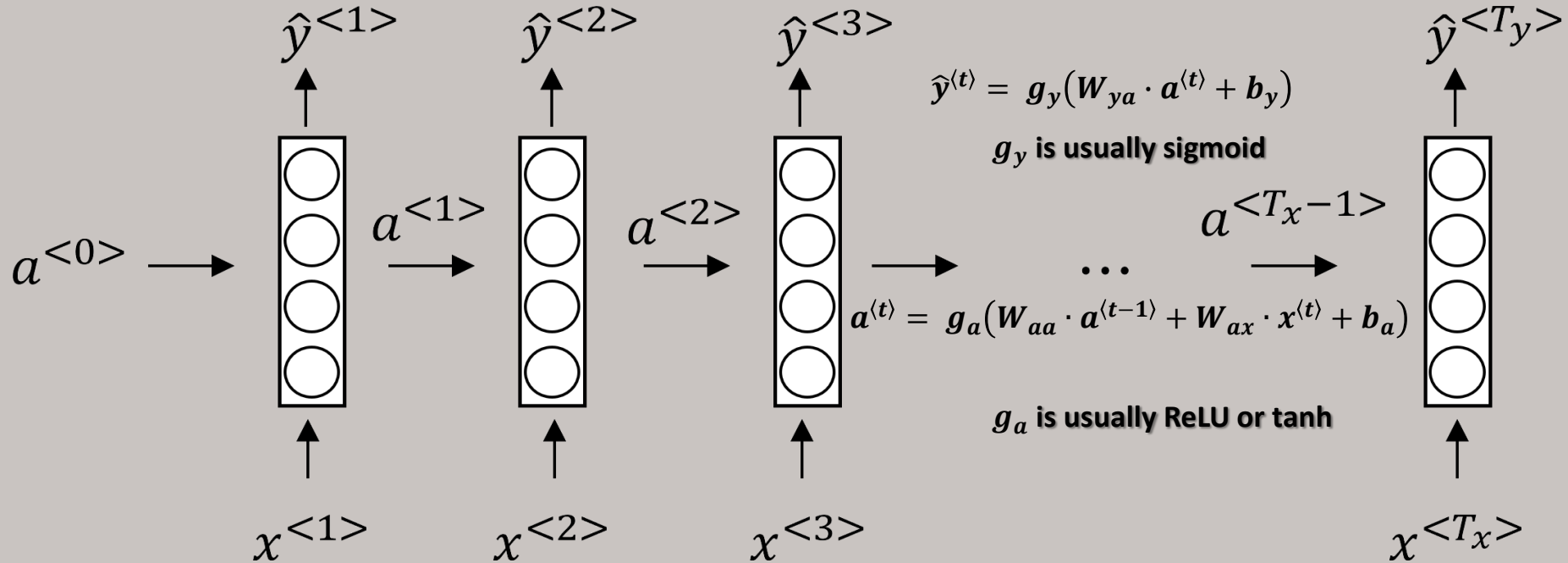
The standard networks that require to associate inputs with features will not work!

We need to find another neural network structure that can work with sequences of inputs (e.g. words) that can move the position in the sequences and take into account the context of previous inputs (e.g. words).

We will use recurrent neural networks



We will use **recurrent neural networks** to overcome the presented difficulties and to allow the network to share features and weights and use the context of previous sequence elements:



In the above network, we put the subsequent elements (e.g. words) on the inputs of the subnetworks which share weights with the other subnetworks (in a nutshell, all these subnetworks are the same network), so the position of the element (word) in the sequence can be different without harm in the representation of this word by the neural network.

Thanks to the connections to the next subnetwork, we can use the context of the processed, previous elements (words) represented by the outputs of previous subnetworks $a^{<t>}$.

Before we start, we should introduce what sequences or sequential data are in detail.

Simplification of the notation



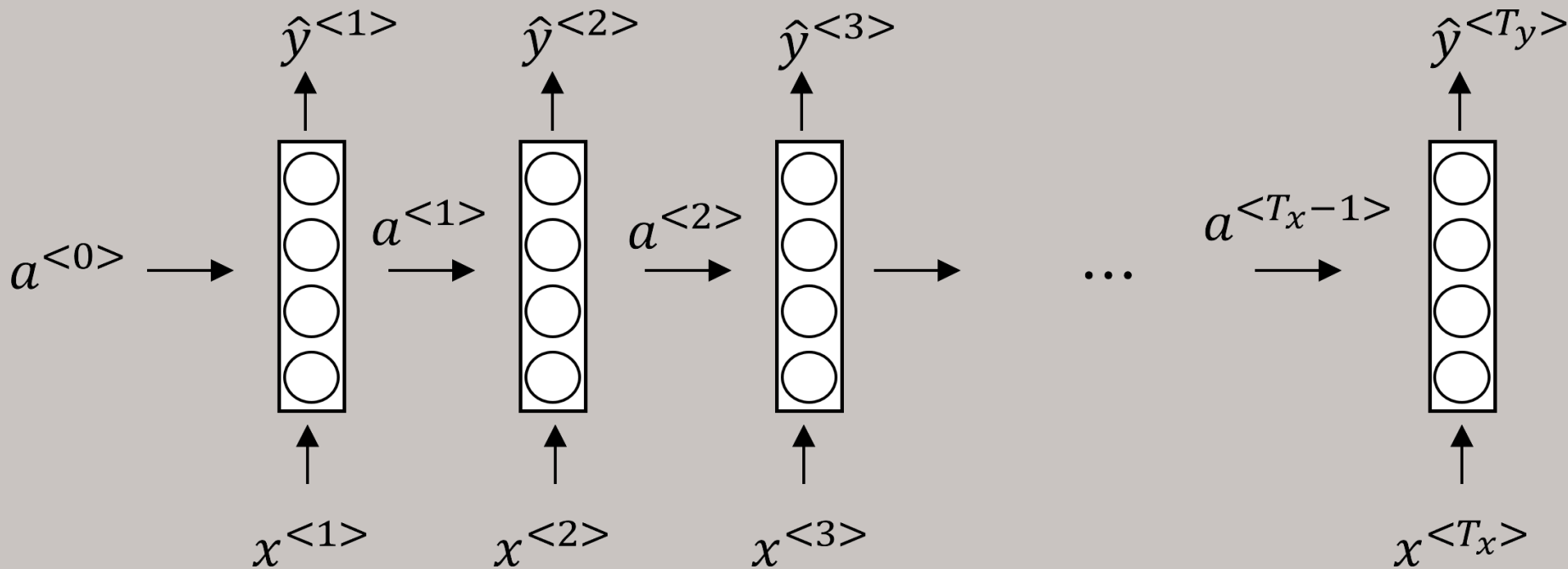
We often use a simplified notation to compute $a^{(t)}$ and $\hat{y}^{(t)}$ which **stacks the weight matrices** and also speed up computations a bit because we do not need to operate on two matrices and adding the multiplication results when computing $a^{(t)}$ but multiplying only once in parallel:

$$a^{(t)} = g_a(W_{aa} \cdot a^{(t-1)} + W_{ax} \cdot x^{(t)} + b_a) = g_a(W_a \cdot [a^{(t-1)}, x^{(t)}] + b_a)$$

g_a is usually ReLU or tan

$$\hat{y}^{(t)} = g_y(W_{ya} \cdot a^{(t)} + b_y) = g_y(W_y \cdot a^{(t)} + b_y)$$

g_y is usually sigmoid



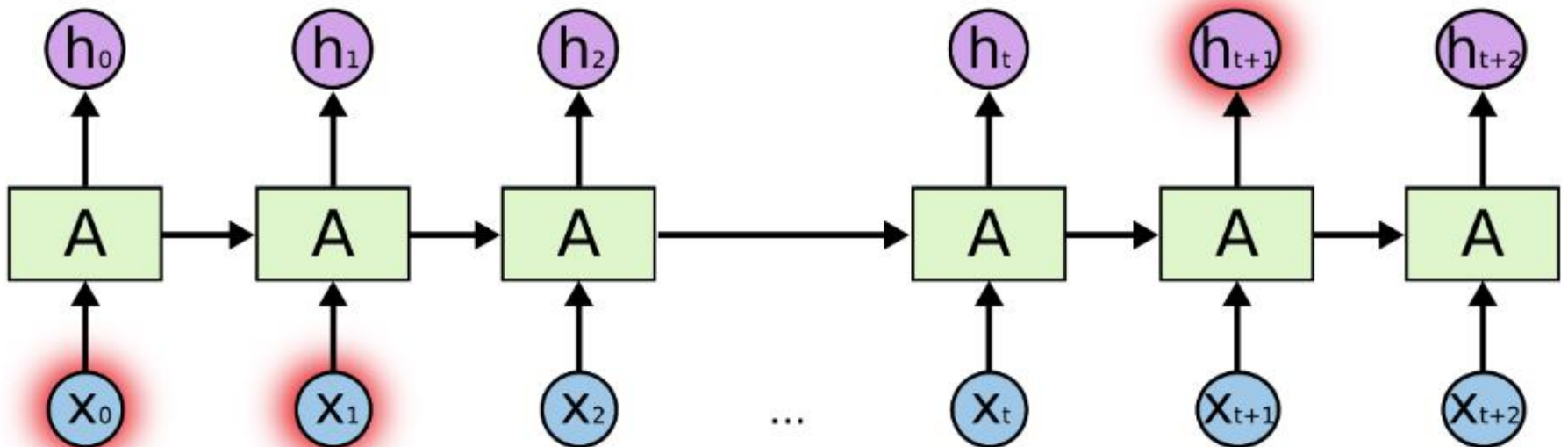
Prediction of Sequence Elements



We can try to predict a next word in a sentence, more generally, a next element in a sequence, we usually use a few previous words, e.g.:

„I grew up in England. Thanks to it, I speak fluent” (English)

RNNs are capable of handling such long-term dependencies.

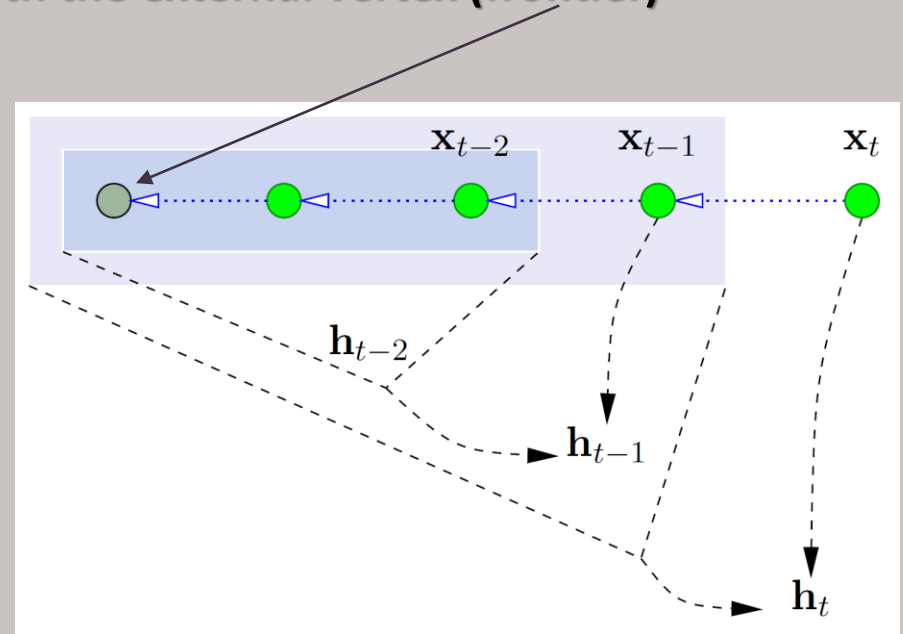


State Transition Function

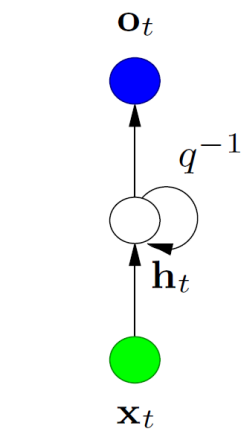
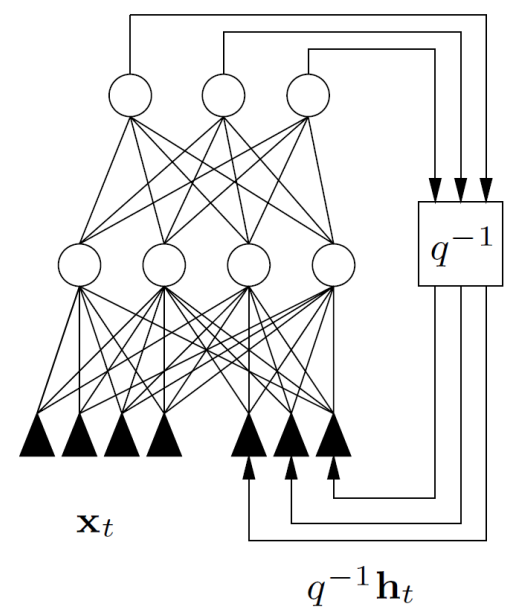


The state transition function defining a single time step can be defined by the shift operator q^{-1} :

- h_0 – an initial step (at $t=0$) associated with the external vertex (frontier)
- $h_t = f(h_{t-1}, x_t)$ – t-step
- $q^{-1} h_t = h_{t-1}$ – unitary time delay
- o_t – output (predicted value)

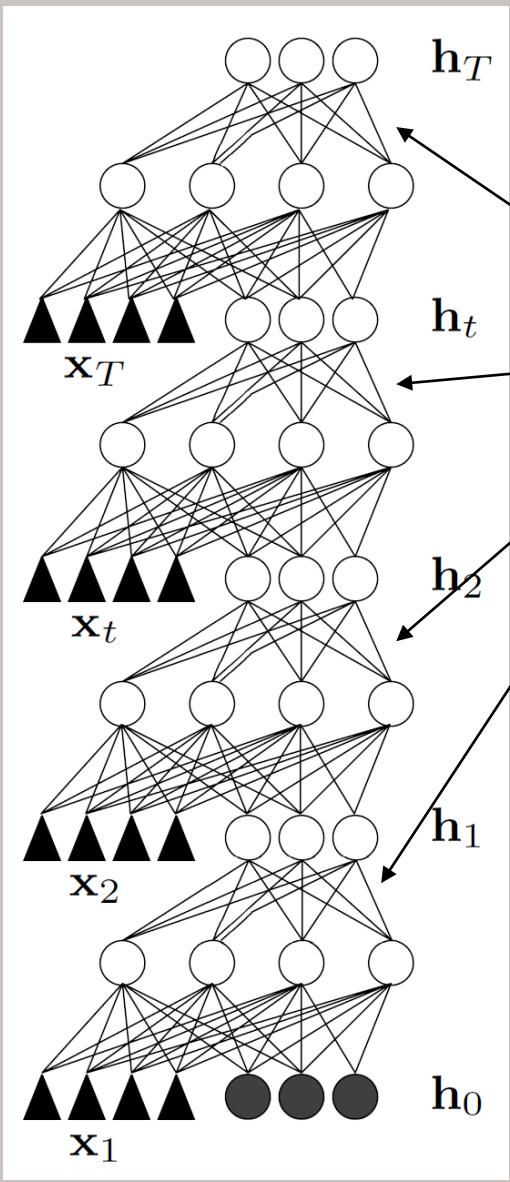


$$h_t = f(h_{t-1}, x_t)$$



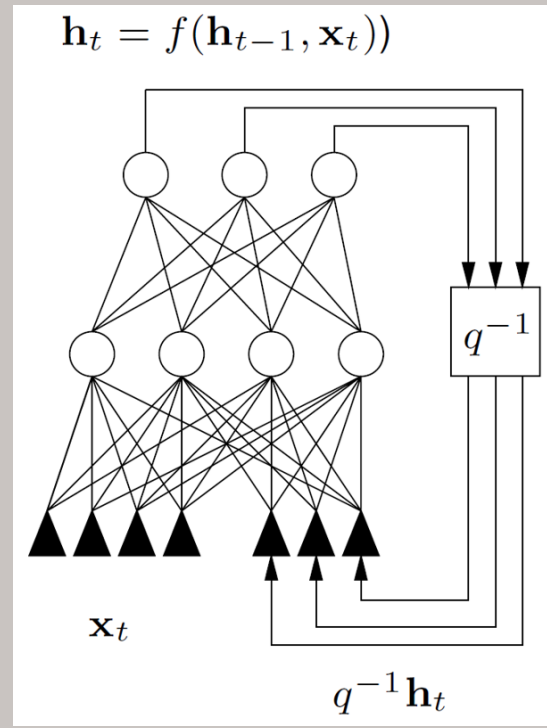
Recursive network

Unfolding Time and Next Sequence Elements



The sequence can be modeled by a deep feedforward neural network which weights can be computed using backpropagation:

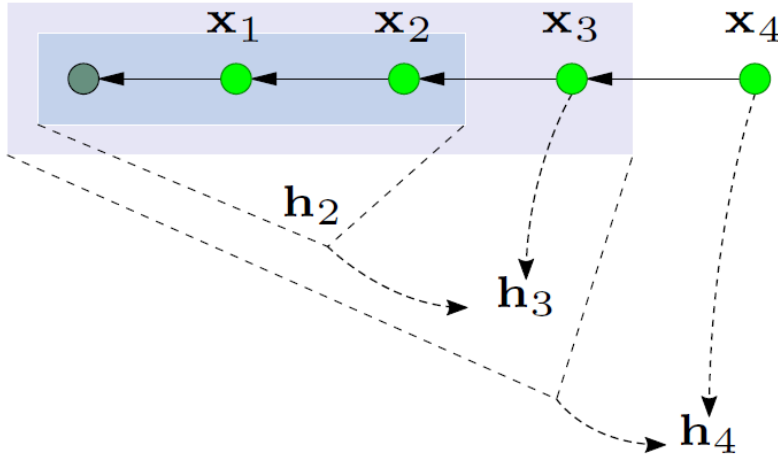
- h_t – is the last state of the whole sequence,
- w – weights are shared between layers (are replicated, the same).



Encoding Networks



For a given sequence s , the encoding network associated to s is formed by unrolling (time unfolding) the recursive network through the input sequence s :



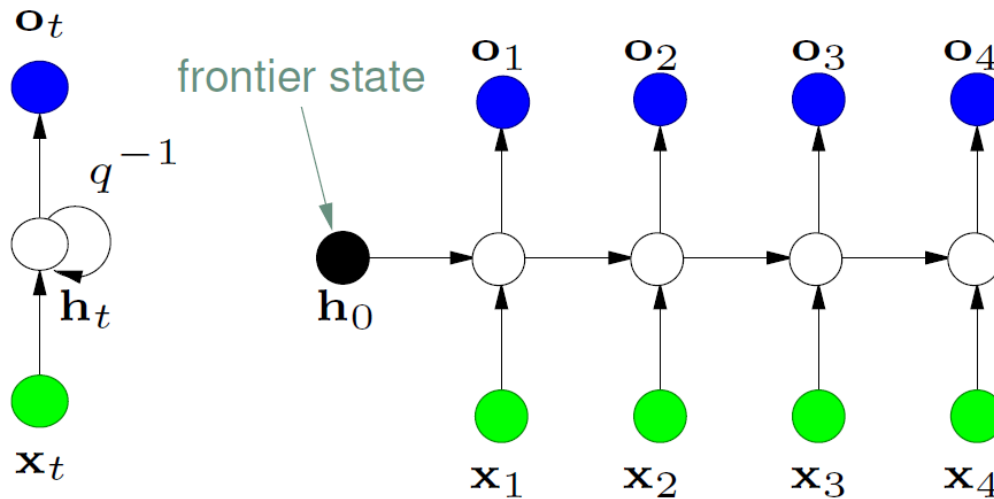
Recursive state update scheme

In linear dynamical systems we can define:

$$\underbrace{h_t}_{\text{state}} = \underbrace{A}_{\text{input weights}} \underbrace{x_t}_{\text{input}} + \underbrace{B}_{\text{hidden weights}} h_{t-1}$$

$$\underbrace{o_t}_{\text{output}} = \underbrace{C}_{\text{output weights}} h_t$$

$$h_0 = 0$$



Recursive network

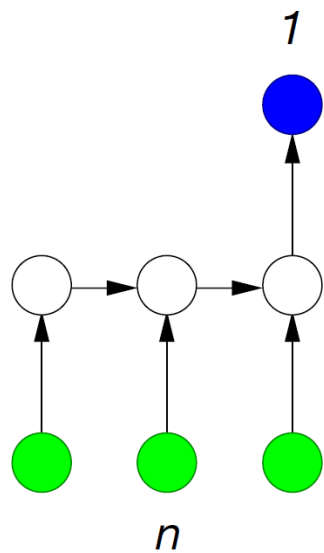
Encoding network

Variety of Sequential Transductions

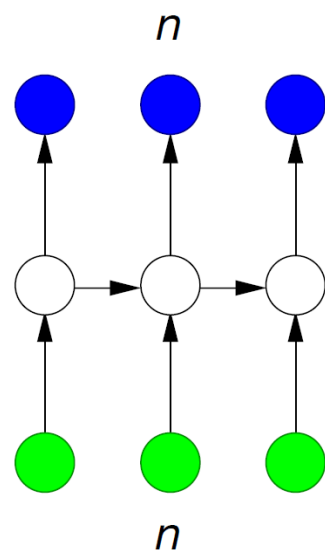


Due to the solved task, we can distinguish various unfolded network structures for:

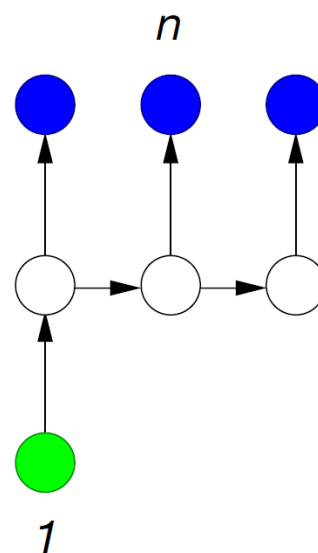
- Sequence classification (e.g. sentiment classification)
- IO transduction (conversion, transfer)
- Sequence generation (e.g. music generation)
- Sequence transduction (from one to another, e.g. sequence translation)



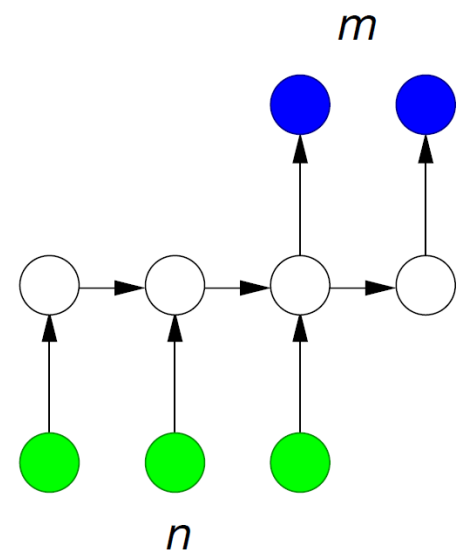
sequence
classification



IO-transduction



sequence
generation

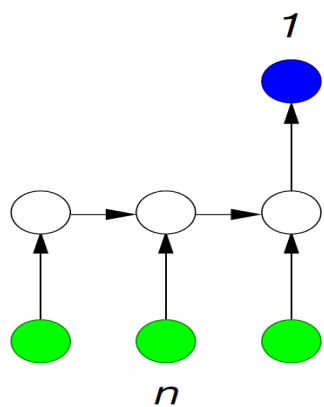


sequence transduction
(in general $n \neq m$)

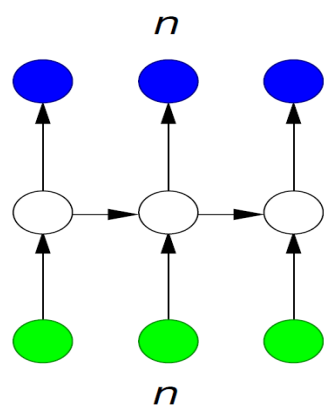
Unification of Various Sequence Tasks



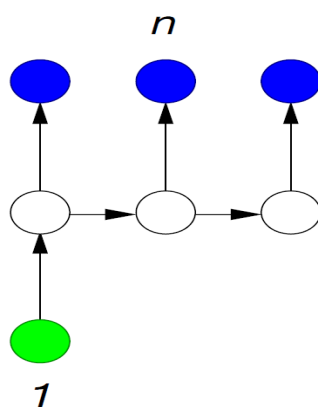
We can easily unify all the presented tasks:



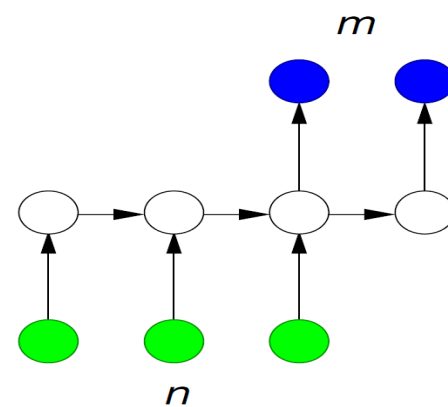
sequence classification



IO-transduction



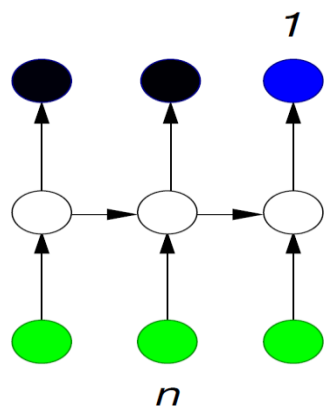
sequence generation



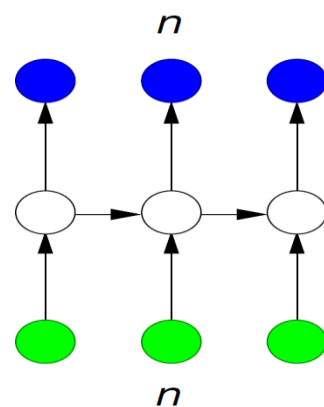
sequence transduction (in general $n \neq m$)



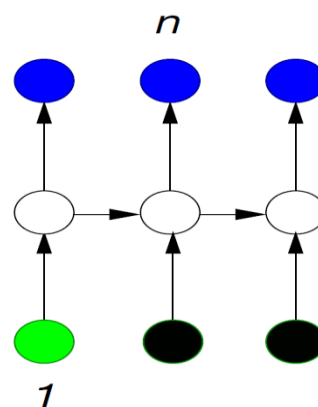
● *don't care input/output*



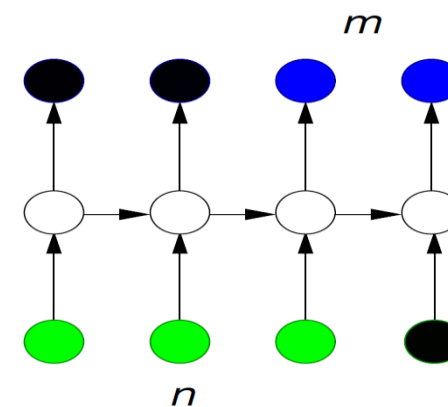
sequence classification



IO-transduction



sequence generation



sequence transduction (in general $n \neq m$)

Shallow Recurrent Neural Networks

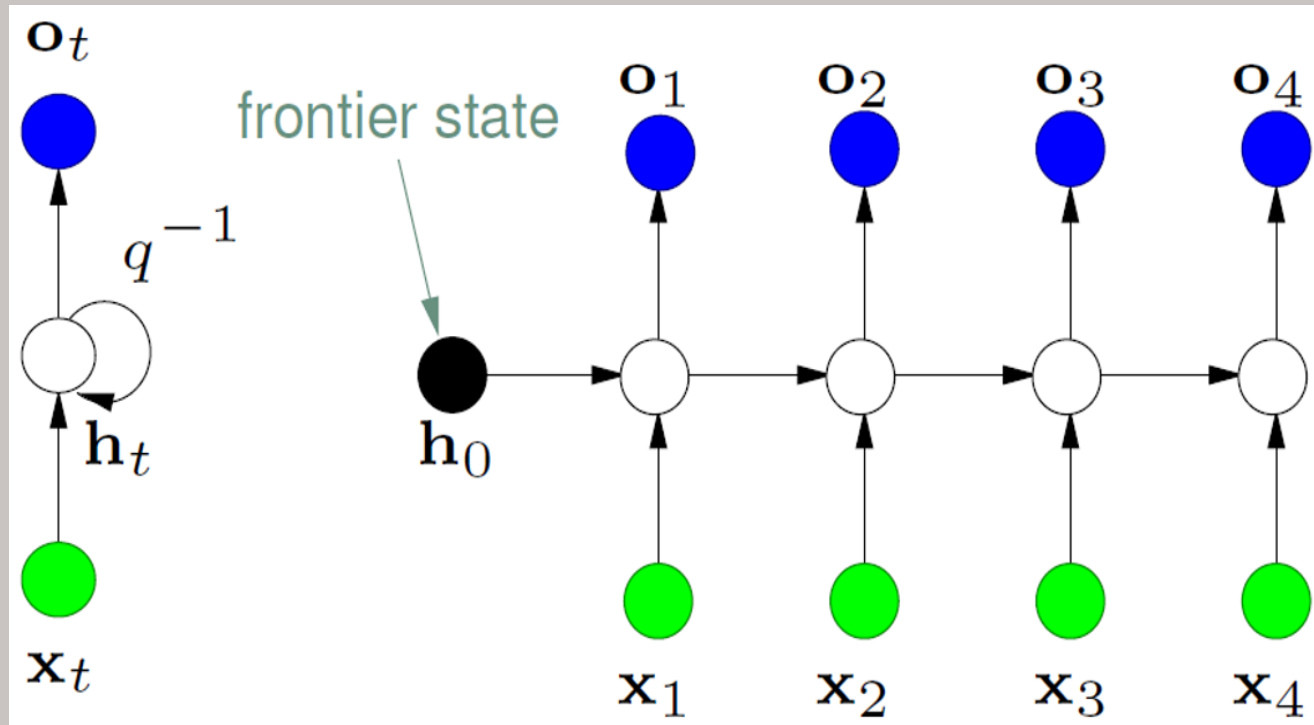


A shallow Recurrent Neural Network (RNN) defines a non-linear dynamical system:

$$\mathbf{h}_t = f(\mathbf{A} \mathbf{x}_t + \mathbf{B} \mathbf{h}_{t-1})$$

$$\mathbf{o}_t = g(\mathbf{C} \mathbf{h}_t)$$

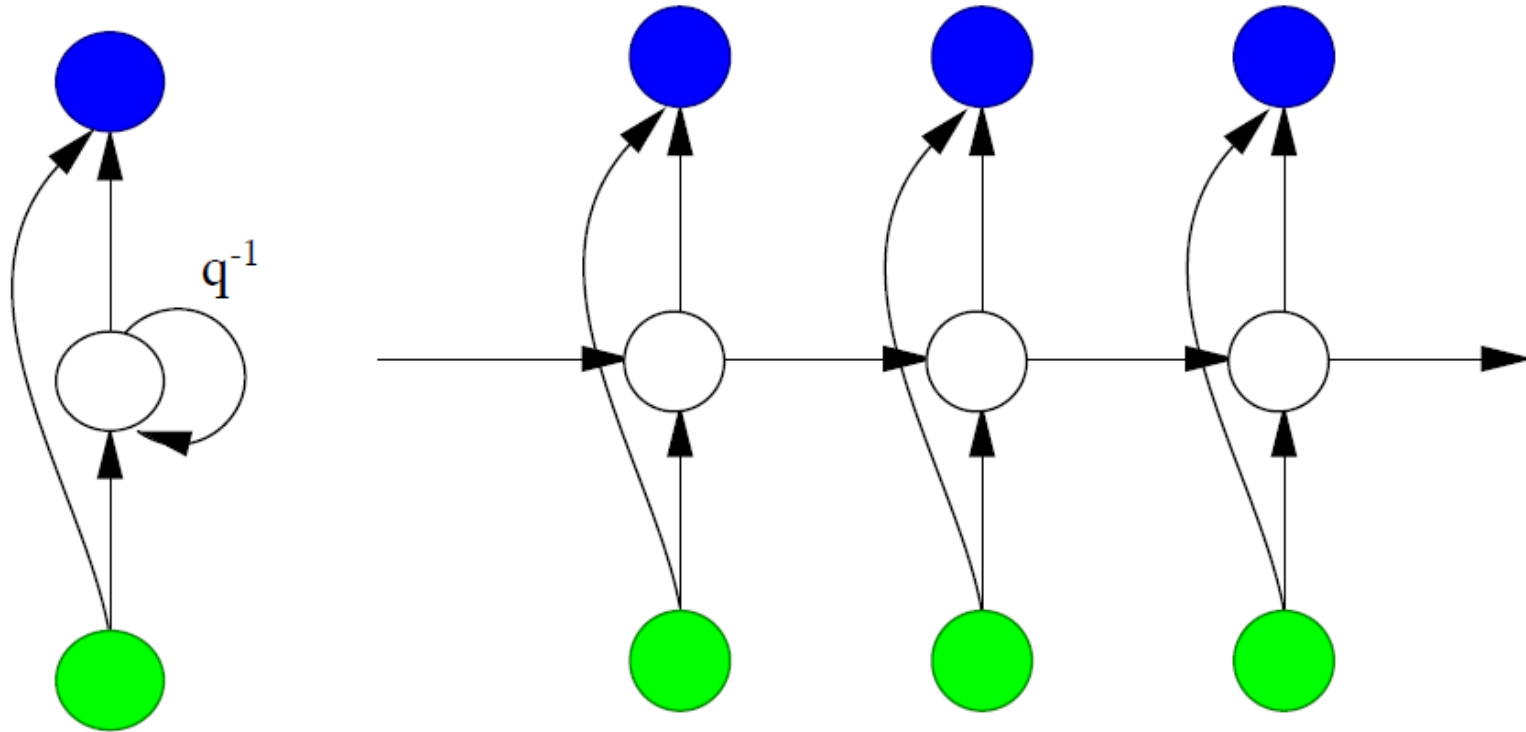
where the functions f and g are non-linear functions (e.g. \tanh), and $\mathbf{h}_0 = 0$ or can be learned jointly with the other parameters.



Additional Architectural Features of RNN



We can use additional short-cut connections between inputs and outputs:

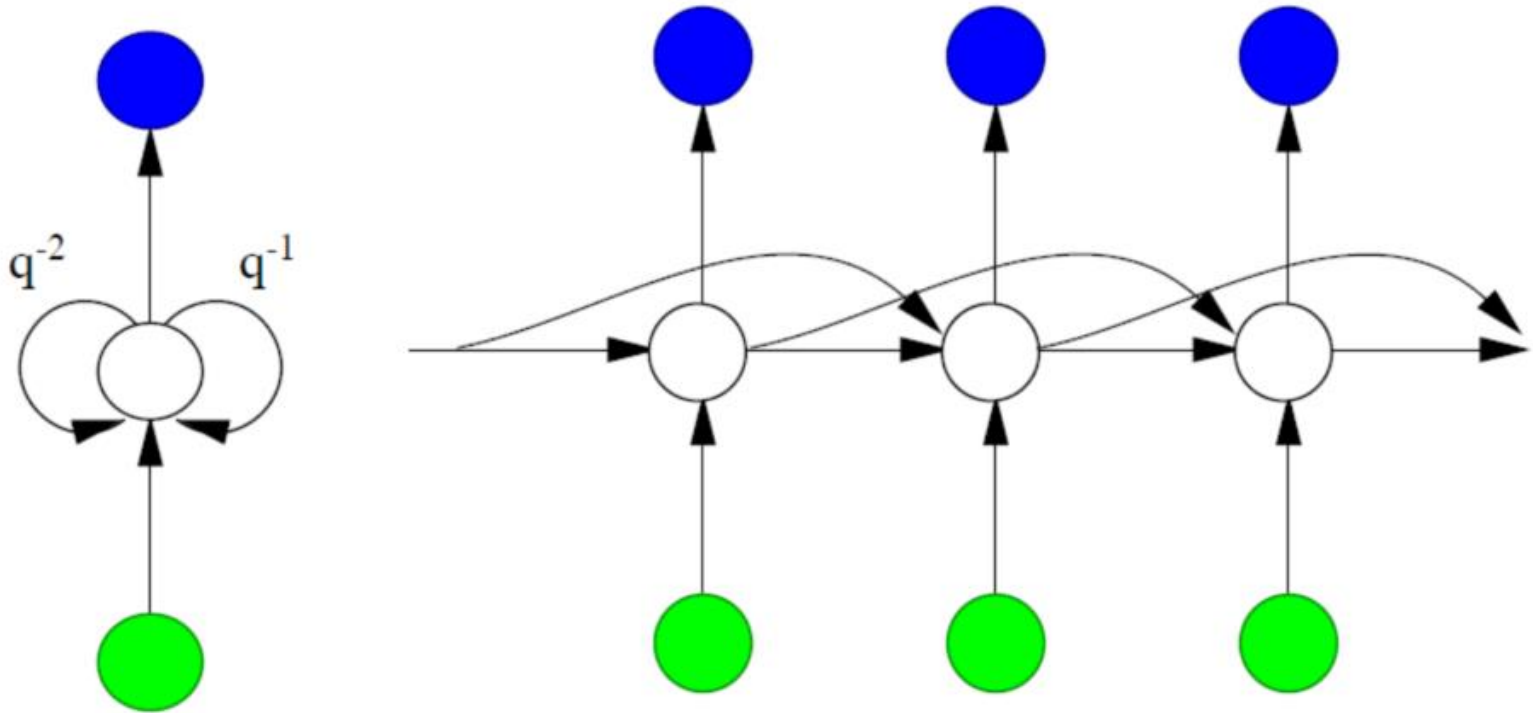


$$\mathbf{o}_t = g(\mathbf{C} \mathbf{h}_t + \mathbf{D} \mathbf{x}_t)$$

Additional Architectural Features of RNN



We can use higher-order states and connections between them, e.g. the 2nd order states:

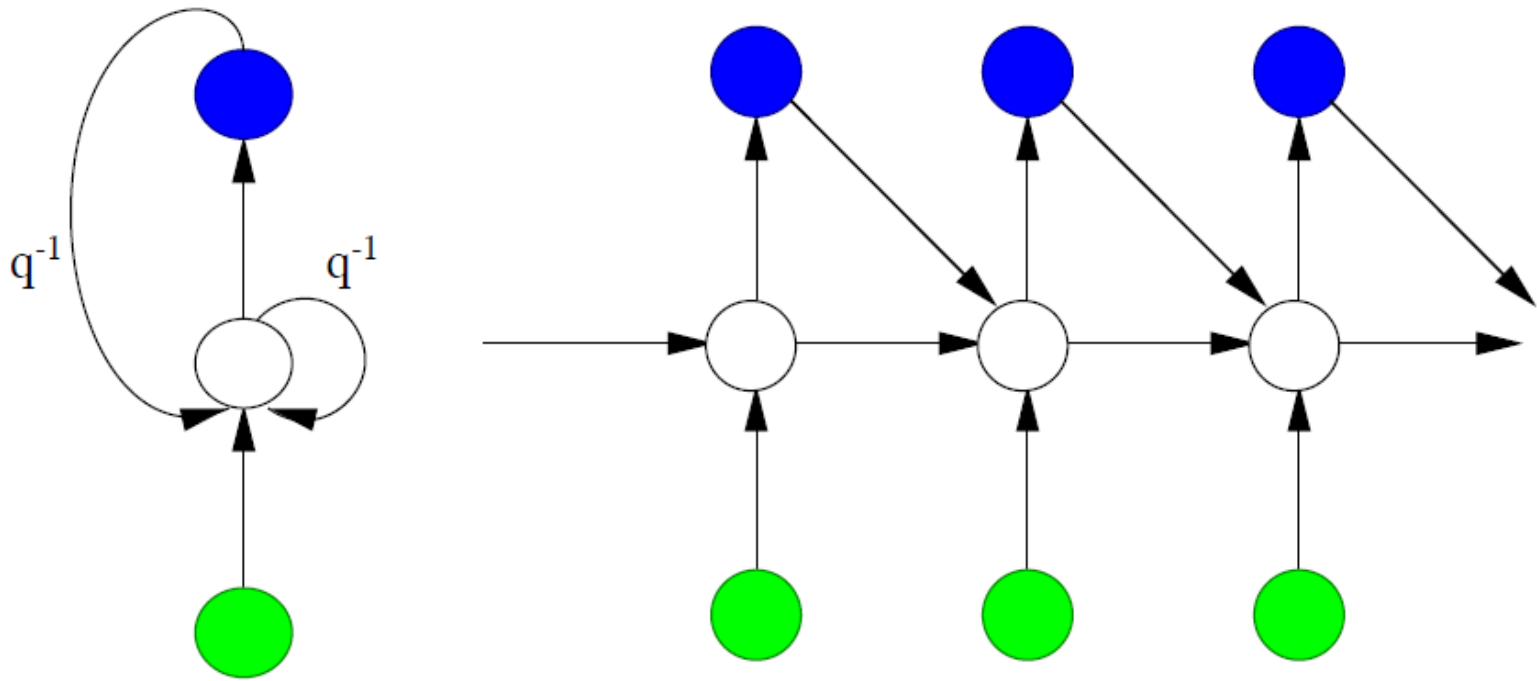


$$\mathbf{h}_t = f(\mathbf{A} \mathbf{x}_t + \mathbf{B}^1 \mathbf{h}_{t-1} + \mathbf{B}^2 \mathbf{h}_{t-2})$$

Additional Architectural Features of RNN



We can use the output to convey contextual information of the next state:

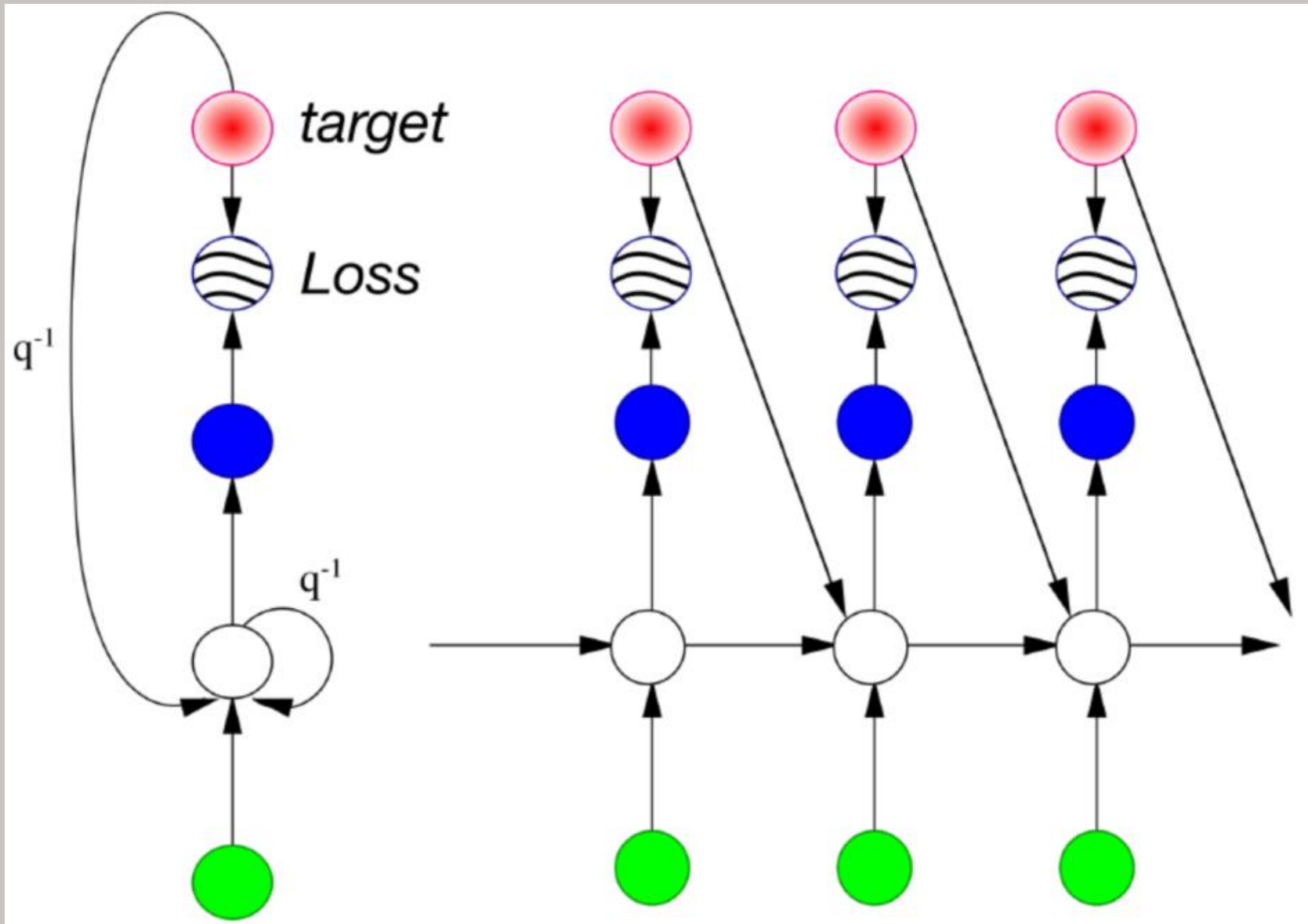


$$\mathbf{h}_t = f(\mathbf{A} \mathbf{x}_t + \mathbf{B}^i \mathbf{h}_{t-1} + \mathbf{B}^o \mathbf{o}_{t-1})$$

Additional Architectural Features of RNN



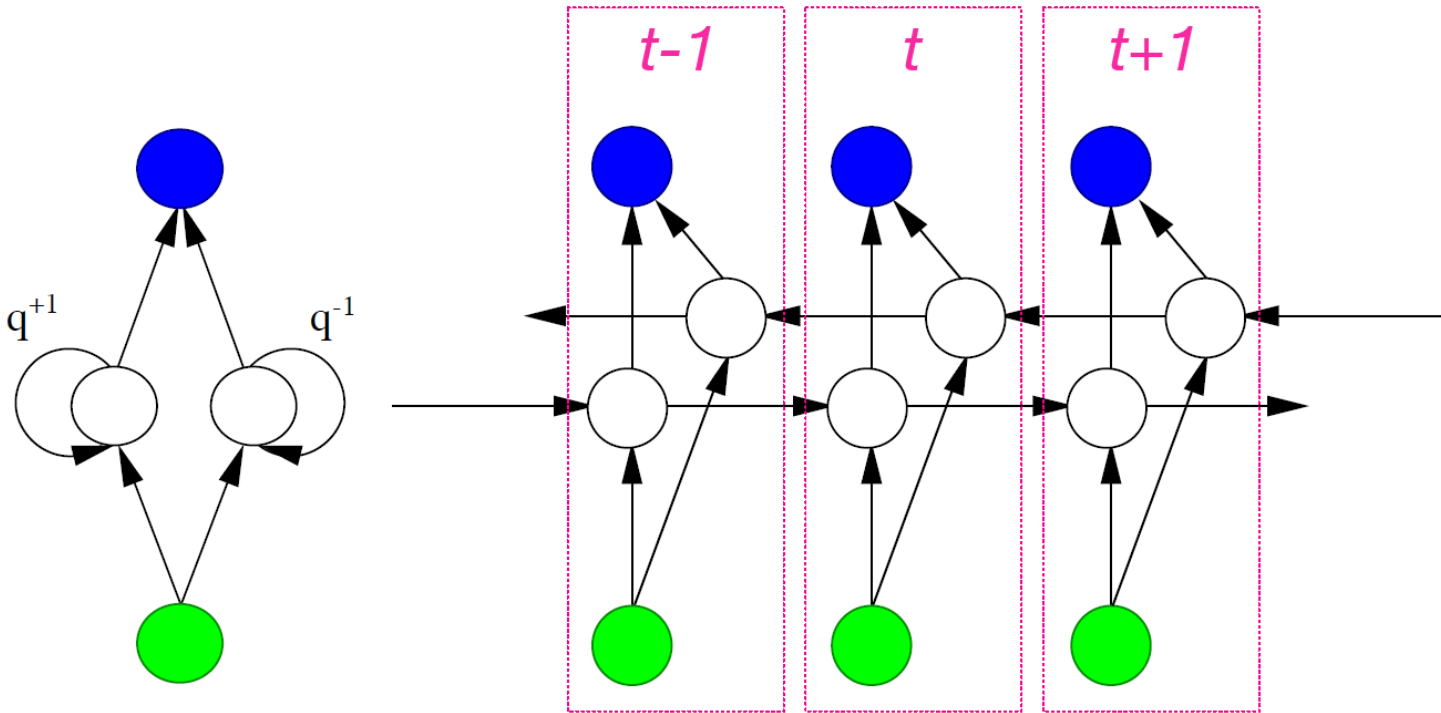
We can also force the target signal (presented by a teacher):



Additional Architectural Features of RNN



We can create Bidirectional Recurrent Neural Networks (BRNN) for off-line processing or when the sequences are not temporal to predict not only next but also previous sequence elements:



$$\mathbf{o}_t = g(\mathbf{C}^p \mathbf{h}_t^p + \mathbf{C}^f \mathbf{h}_t^f)$$

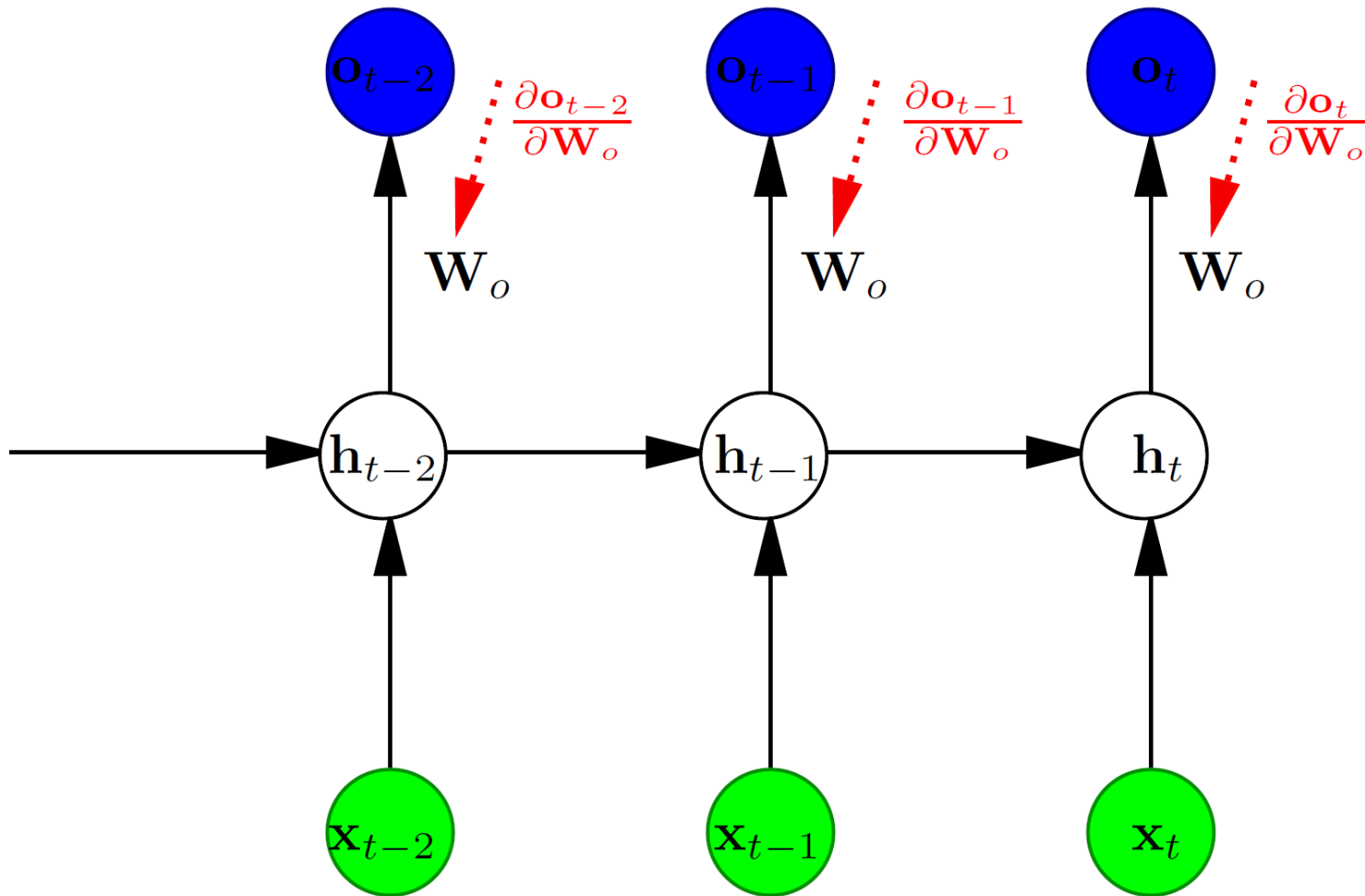
$$\mathbf{h}_t^p = f(\mathbf{A}^p \mathbf{x}_t + \mathbf{B}^p \mathbf{h}_{t-1})$$

$$\mathbf{h}_t^f = f(\mathbf{A}^f \mathbf{x}_t + \mathbf{B}^f \mathbf{h}_{t+1})$$

Back-Propagation Through Time (BPTT)



The backpropagation algorithm can be adapted to sequential patterns:

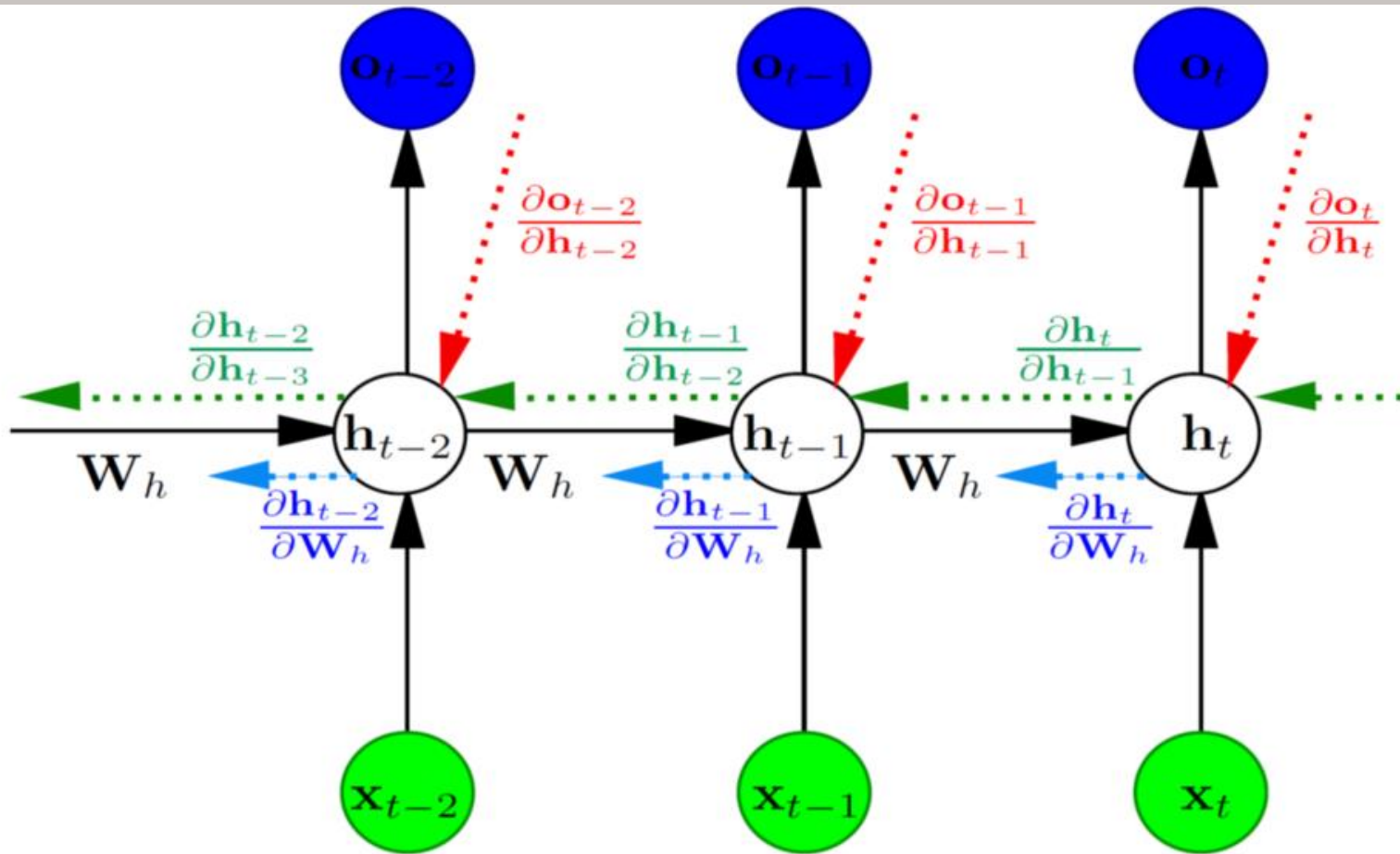


$$\frac{\partial \mathbf{E}}{\partial \mathbf{W}_o} = \dots + \frac{\partial \mathbf{E}_{t-2}}{\partial \mathbf{o}_{t-2}} \frac{\partial \mathbf{o}_{t-2}}{\partial \mathbf{W}_o} + \frac{\partial \mathbf{E}_{t-1}}{\partial \mathbf{o}_{t-1}} \frac{\partial \mathbf{o}_{t-1}}{\partial \mathbf{W}_o} + \frac{\partial \mathbf{E}_t}{\partial \mathbf{o}_t} \frac{\partial \mathbf{o}_t}{\partial \mathbf{W}_o}$$

Back-Propagation Through Time (BPTT)



The backpropagation algorithm can be adapted to sequential patterns:

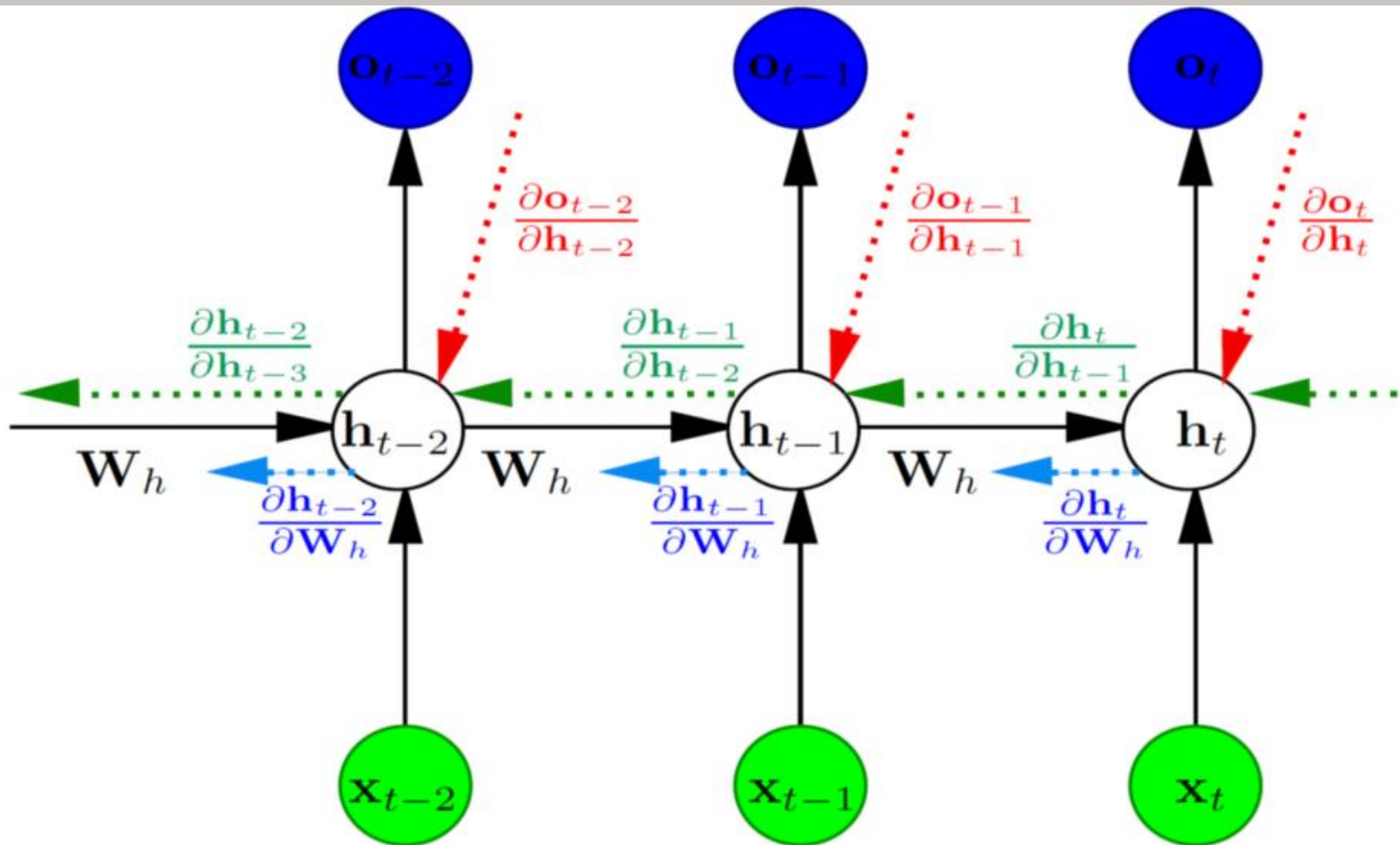


$$\frac{\partial o_t}{\partial W_h} = \sum_{t'=1}^t \frac{\partial o_t}{\partial h_t} \frac{\partial h_t}{\partial h_{t'}} \frac{\partial h_{t'}}{\partial W_h}, \text{ where } \frac{\partial h_t}{\partial h_{t'}} = \prod_{j=t'+1}^t \frac{\partial h_j}{\partial h_{j-1}}$$

Back-Propagation Through Time (BPTT)



The backpropagation algorithm can be adapted to sequential patterns:

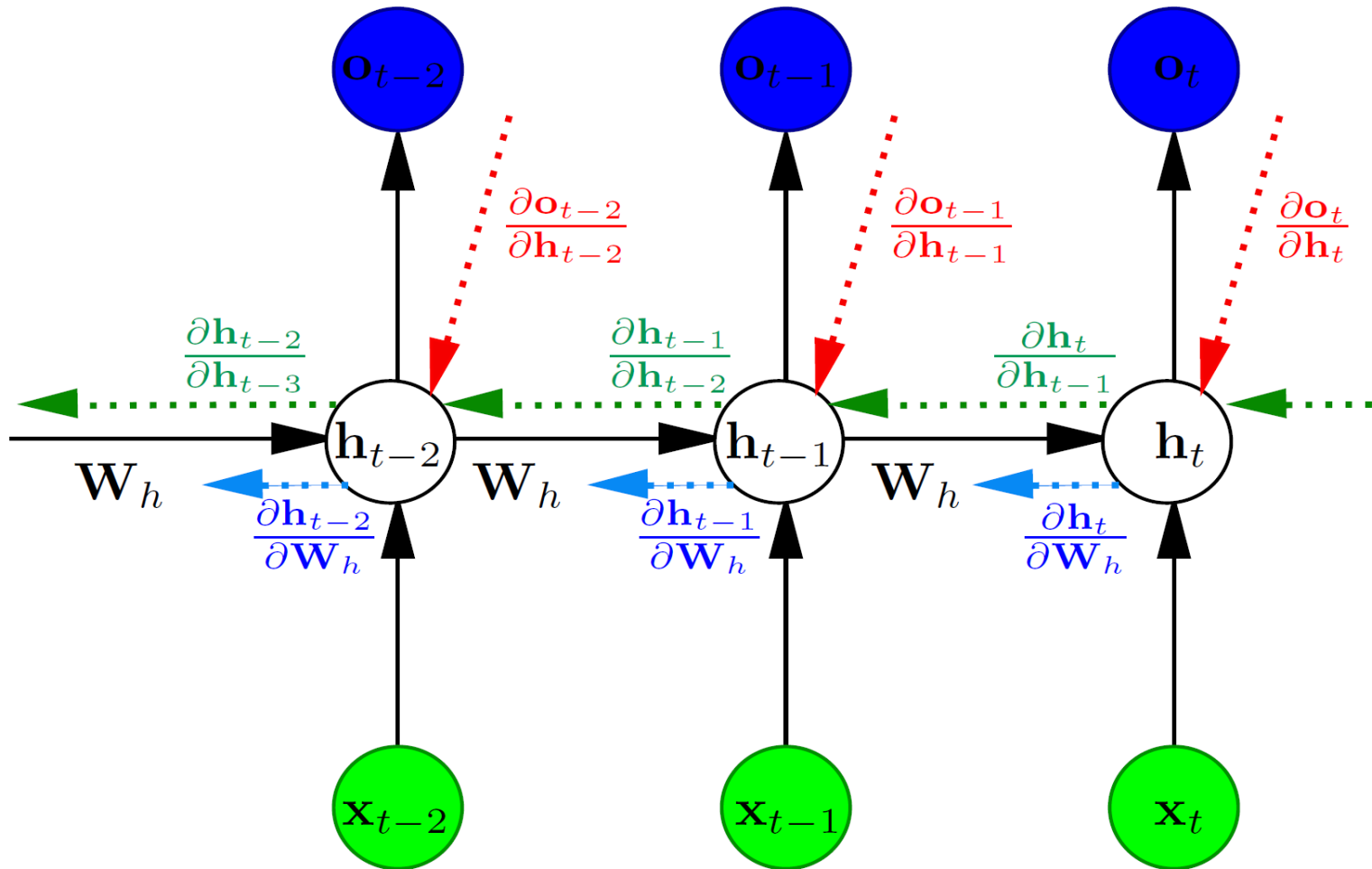


$$\frac{\partial o_t}{\partial \mathbf{W}_h} = \sum_{t'=1}^t \frac{\partial o_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t'}} \frac{\partial \mathbf{h}_{t'}}{\partial \mathbf{W}_h}, \text{ where } \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t'}} = \prod_{j=t'+1}^t \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}$$

Back-Propagation Through Time (BPTT)



The backpropagation algorithm can be adapted to sequential patterns:

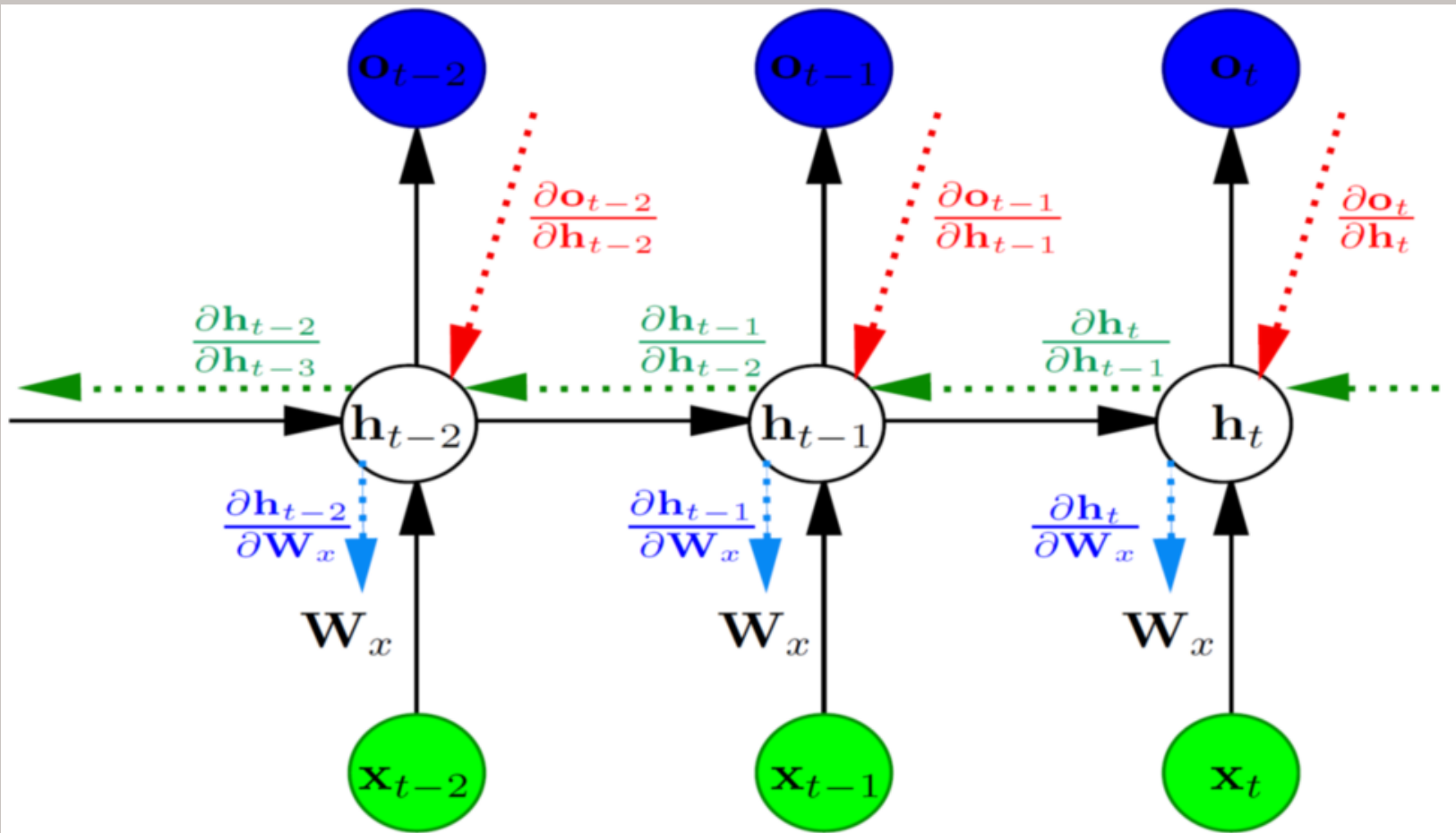


$$\frac{\partial E}{\partial W_h} = \dots + \frac{\partial E_{t-2}}{\partial o_{t-2}} \frac{\partial o_{t-2}}{\partial W_h} + \frac{\partial E_{t-1}}{\partial o_{t-1}} \frac{\partial o_{t-1}}{\partial W_h} + \frac{\partial E_t}{\partial o_t} \frac{\partial o_t}{\partial W_h}$$

Back-Propagation Through Time (BPTT)



The backpropagation algorithm can be adapted to sequential patterns:

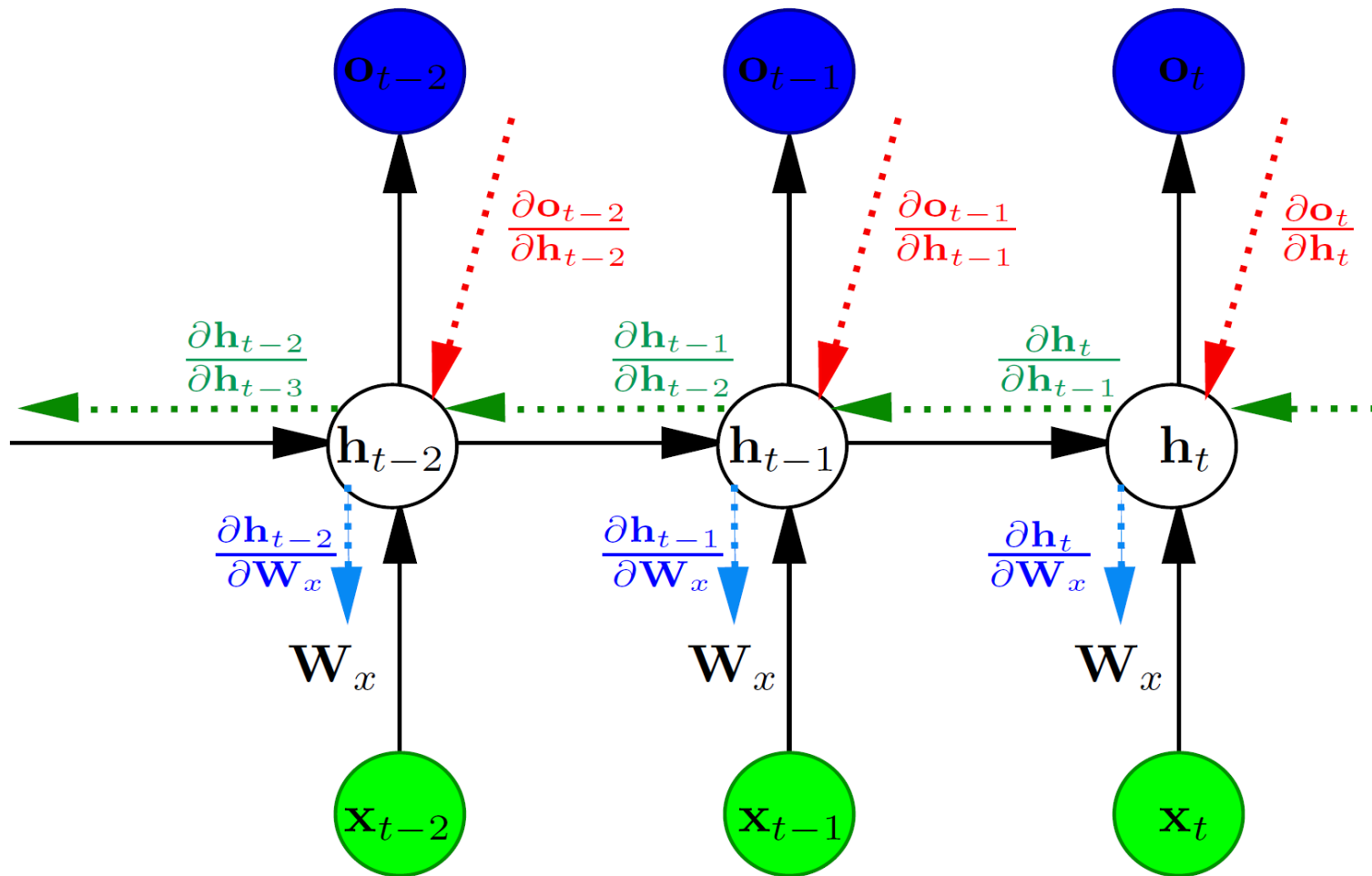


$$\frac{\partial o_t}{\partial W_x} = \sum_{t'=1}^t \frac{\partial o_t}{\partial h_t} \frac{\partial h_t}{\partial h_{t'}} \frac{\partial h_{t'}}{\partial W_x}$$

Back-Propagation Through Time (BPTT)



The backpropagation algorithm can be adapted to sequential patterns:

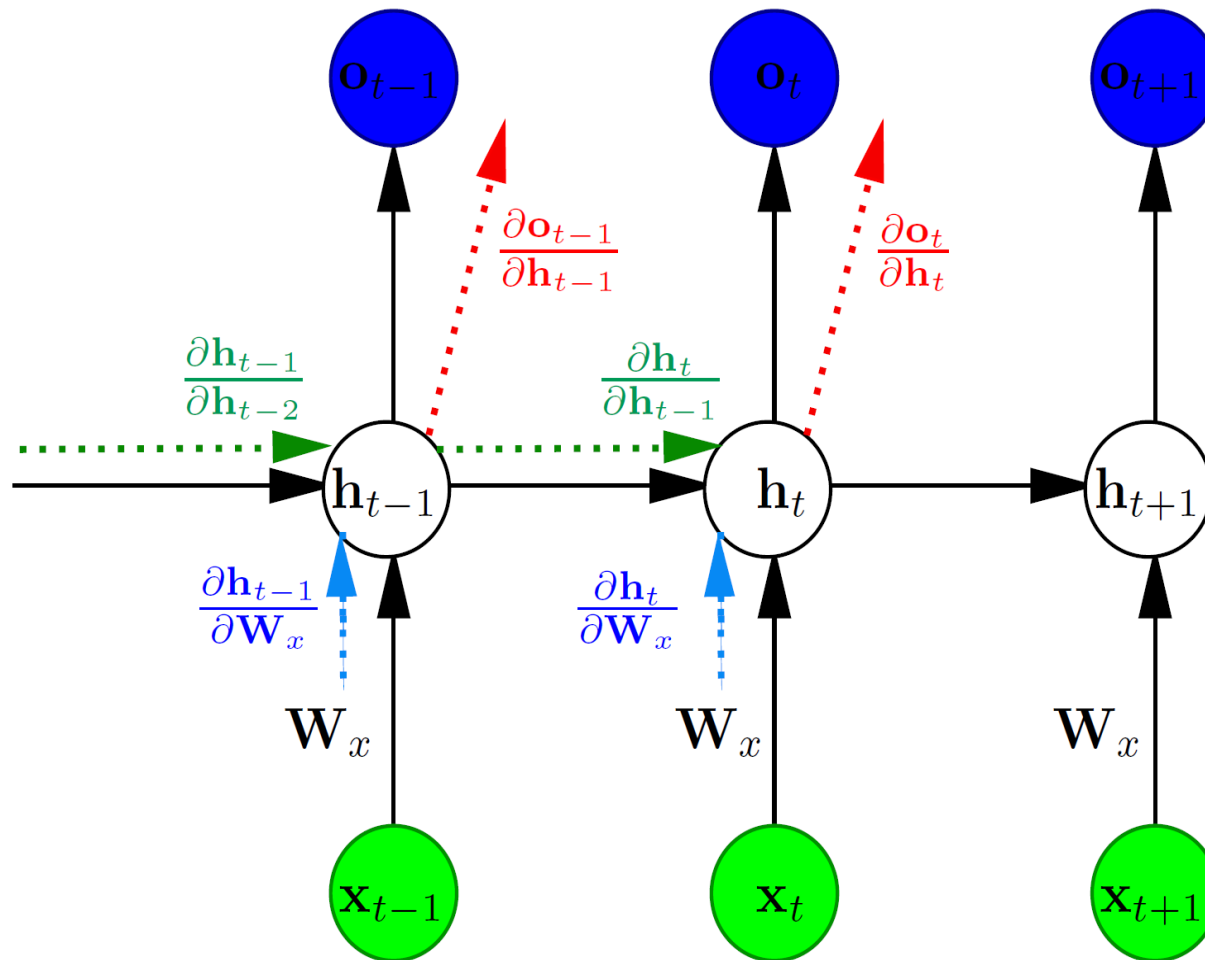


$$\frac{\partial E}{\partial W_x} = \dots + \frac{\partial E_{t-2}}{\partial o_{t-2}} \frac{\partial o_{t-2}}{\partial W_x} + \frac{\partial E_{t-1}}{\partial o_{t-1}} \frac{\partial o_{t-1}}{\partial W_x} + \frac{\partial E_t}{\partial o_t} \frac{\partial o_t}{\partial W_x}$$

Real-Time Recurrent Learning (RTRL)



Real-Time Recurrent Learning (RTRL) adapted to sequential patterns:

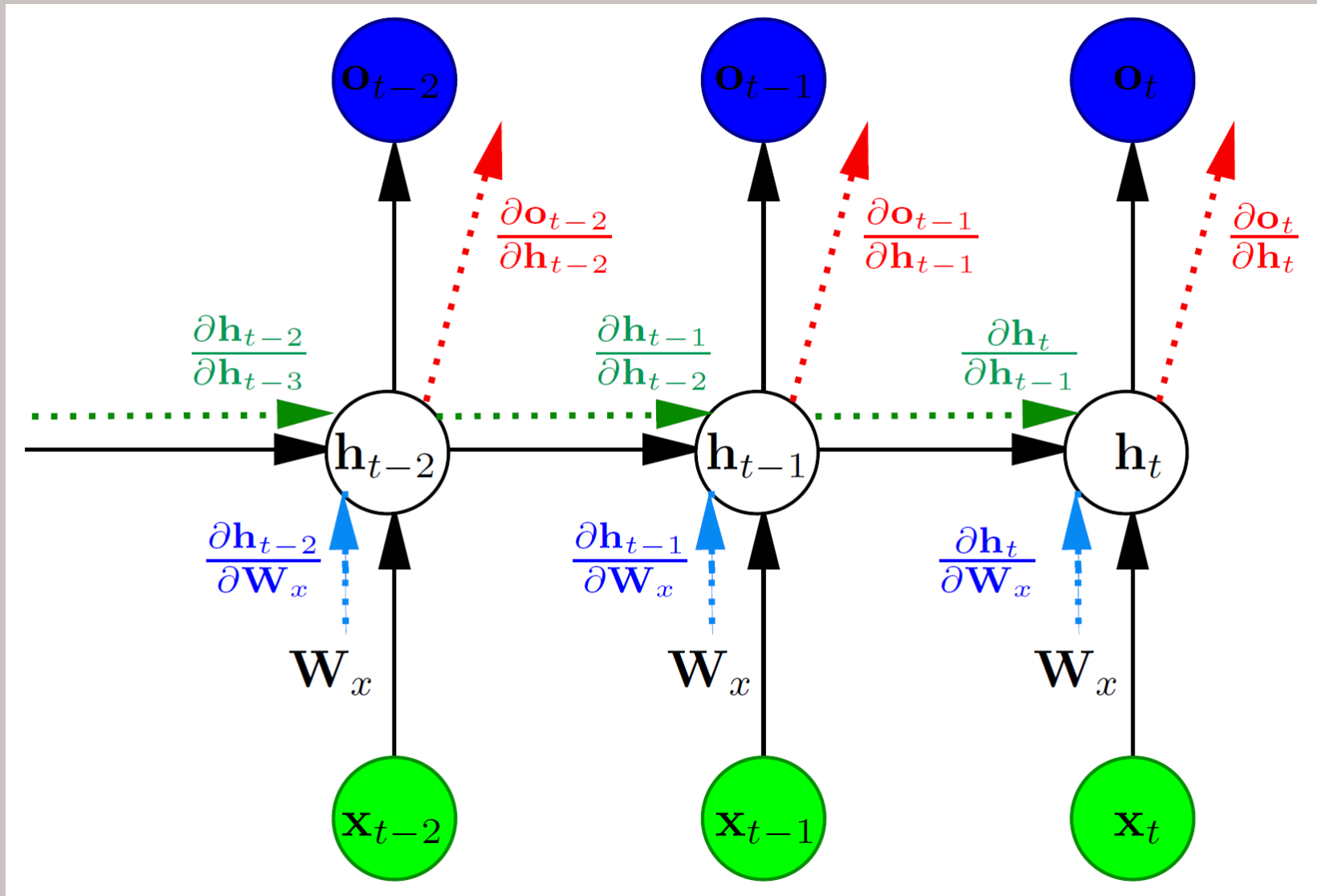


$$\frac{\partial \mathbf{E}|_t}{\partial \mathbf{W}_x} = \dots + \frac{\partial \mathbf{E}_{t-1}}{\partial o_{t-1}} \frac{\partial o_{t-1}}{\partial \mathbf{W}_x} + \frac{\partial \mathbf{E}_t}{\partial o_t} \frac{\partial o_t}{\partial \mathbf{W}_x} = \frac{\partial \mathbf{E}|_{t-1}}{\partial \mathbf{W}_x} + \frac{\partial \mathbf{E}_t}{\partial o_t} \frac{\partial o_t}{\partial \mathbf{W}_x}$$

Real-Time Recurrent Learning (RTRL)



Real-Time Recurrent Learning (RTRL) computes partial derivatives during the forward phase:



Comparison of BPTT and RTRL



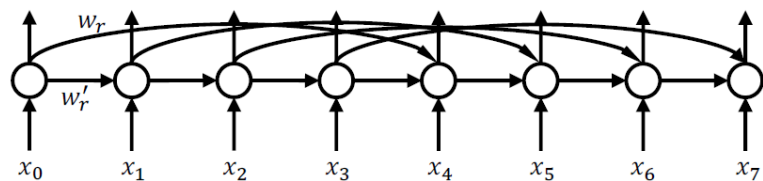
Both BPTT and RTRL compute the same gradients but in different ways.
They differ in computational complexity:

| | <i>Space</i> | <i>Time</i> |
|-------------|--------------|-------------|
| <i>BPTT</i> | $O(NT)$ | $O(N^2T)$ |
| <i>RTRL</i> | $O(N^3)$ | $O(N^4)$ |

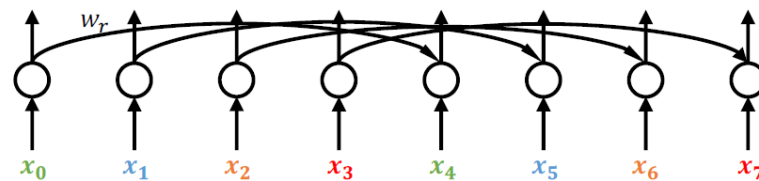
T : time steps

N : number of units

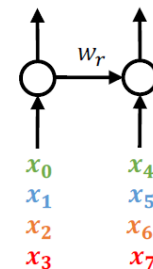
Deep Dilated Recurrent Neural Networks



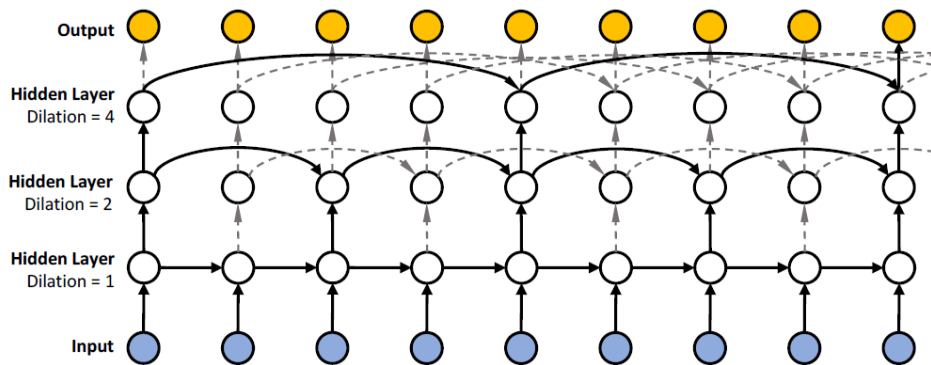
single-layer RNN with recurrent skip connections



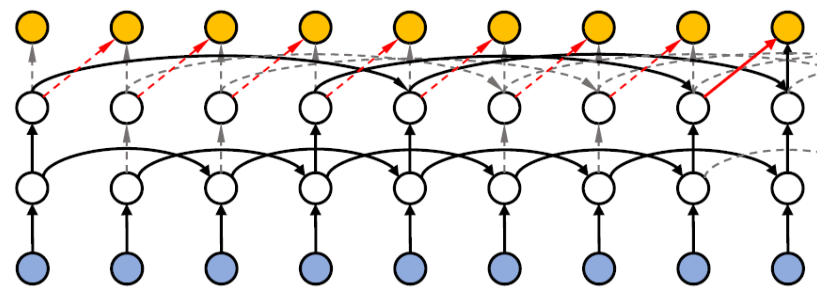
same RNN with dilated recurrent skip connections



Deep Dilated Recurrent Neural Networks



$$s^{(l)} = M^{l-1}, l = 1, \dots, L$$



$$s^{(l)} = M^{(l-1+l_0)}, l = 1, \dots, L \text{ and } l_0 \geq 0$$

convolutional layer is appended as the final layer (red arrows)

Vanishing/Exploding Gradient Problems



In both BPTT and RTRL, we come across exploding and vanishing gradient problems:

Exploding gradients are a problem where large error gradients accumulate and result in very large updates to neural network model weights during training. This effects in instability of the model and difficulty to learn from training data, especially over long input sequences of data.

In order to robustly store past information, the dynamics of the network must exhibit attractors but, in their presence, **gradients vanish** going backward in time, so no learning with gradient descent is possible!

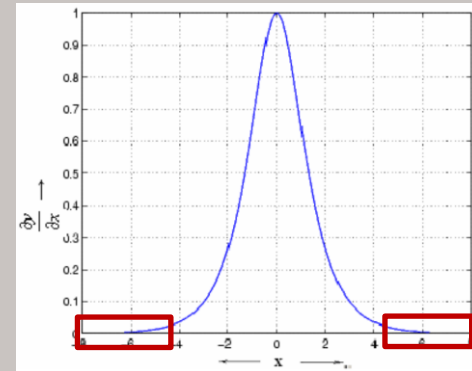
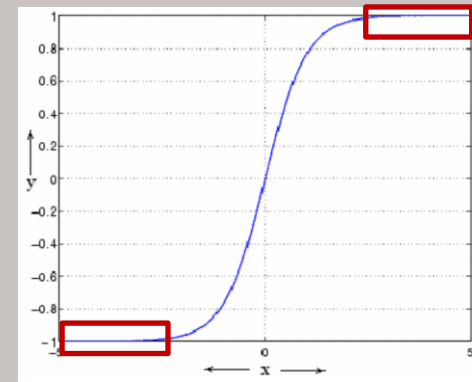
To reduce the vanishing/exploding gradient problems, we can:

Modify or change the architecture or the network model:

- Long Short-Term Memory (LSTM) units
- Reservoir Computing: Echo State Networks and Liquid State Machines

Modify or change the algorithm:

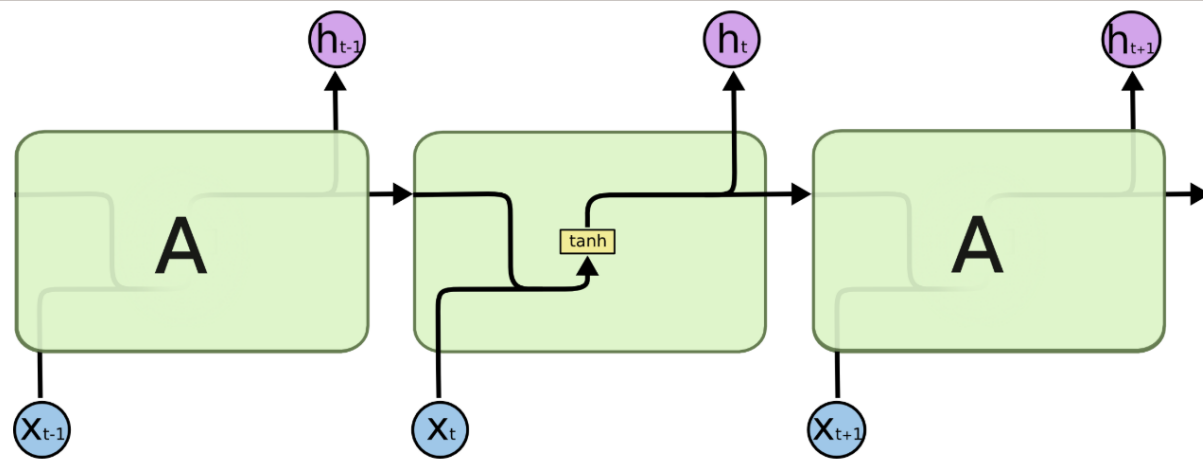
- Hessian Free Optimization
- Smart Initialization: pre-training techniques
- Clipping gradients (check for and limit the size of gradients during the training of the network)
- Truncated Backpropagation through time (updating across fewer prior time steps during training)
- Weight Regularization (apply a penalty to the networks loss function for large weight values)



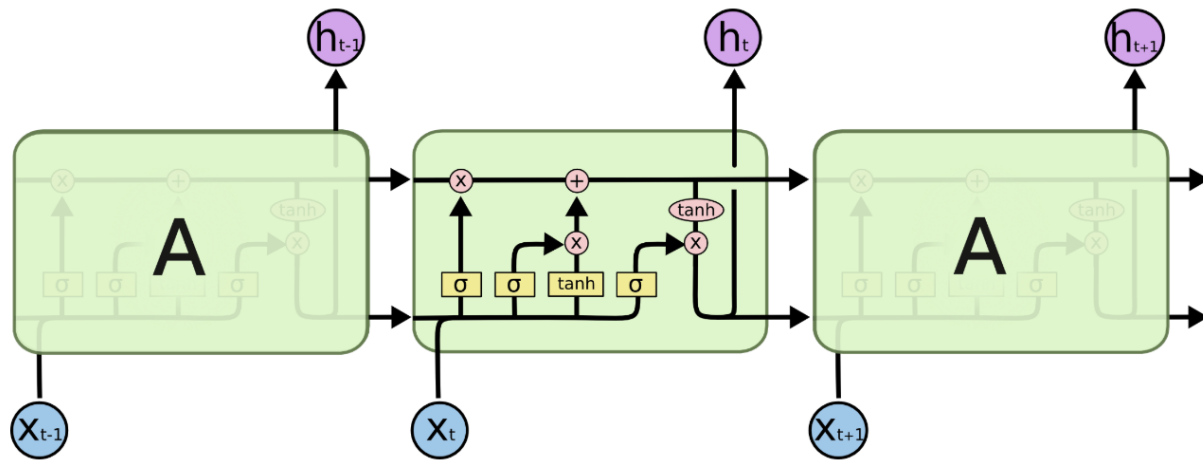
Long Short-Term Memory (LSTM)



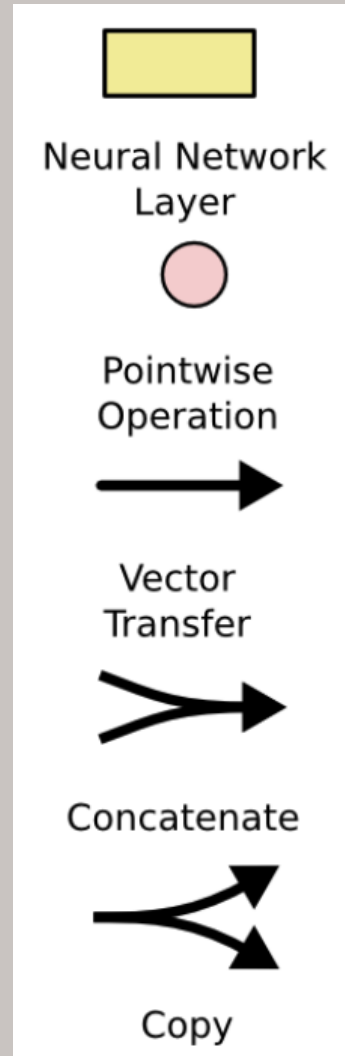
Long Short-Term Memory networks are a special kind of Recurrent Neural Networks, containing four (instead of one) interacting layers and capable of learning long-term dependencies.



The repeating module in a standard RNN contains a single layer.



The repeating module in an LSTM contains four interacting layers.



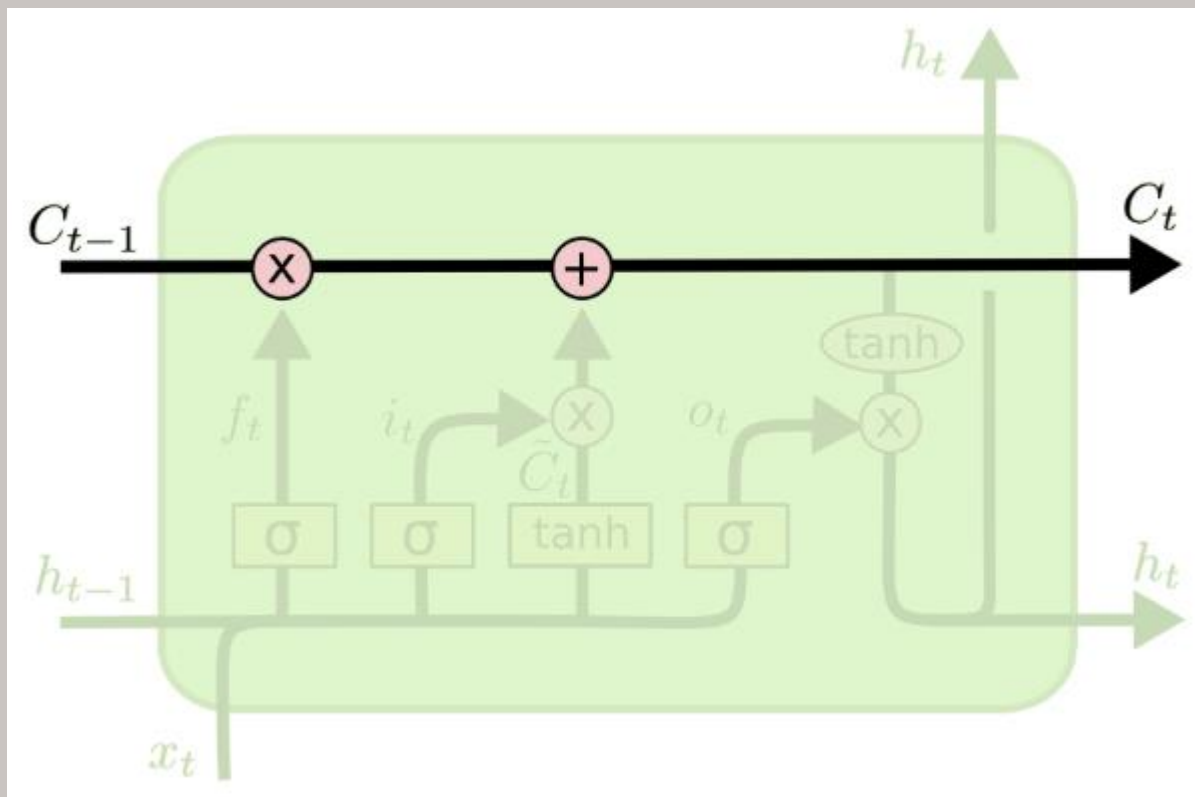
Cell State of LSTMs



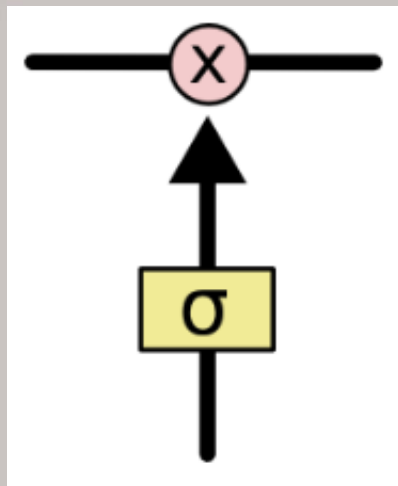
The key to LSTM is the cell state represented by the horizontal line running through the top of the diagram. It is a kind of **conveyor belt**.

It runs straight down the entire chain, with only some minor linear interactions.

The LSTM has the ability to **remove** or **add information to the cell state**, carefully regulated by structures called **gates**.



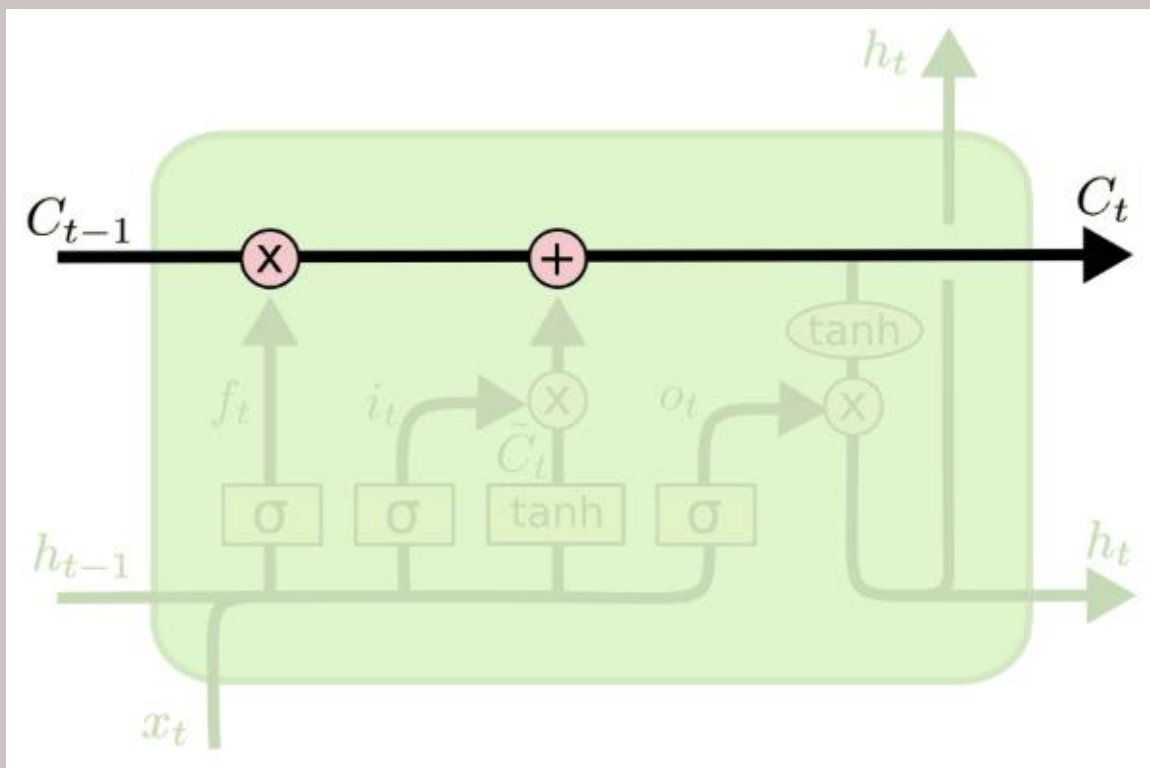
Gates of LSTMs



Gates are a way to optionally let information through.

They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.

The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of **zero** means “**let nothing through,**” while a value of **one** means “**let everything through!**”



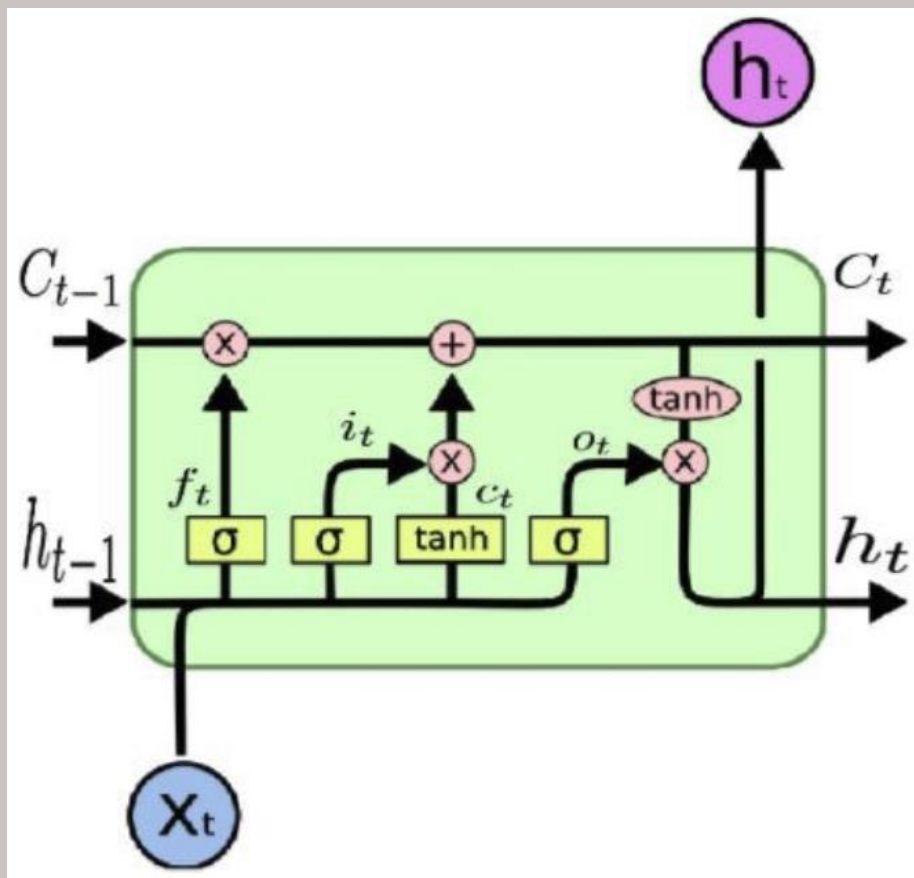
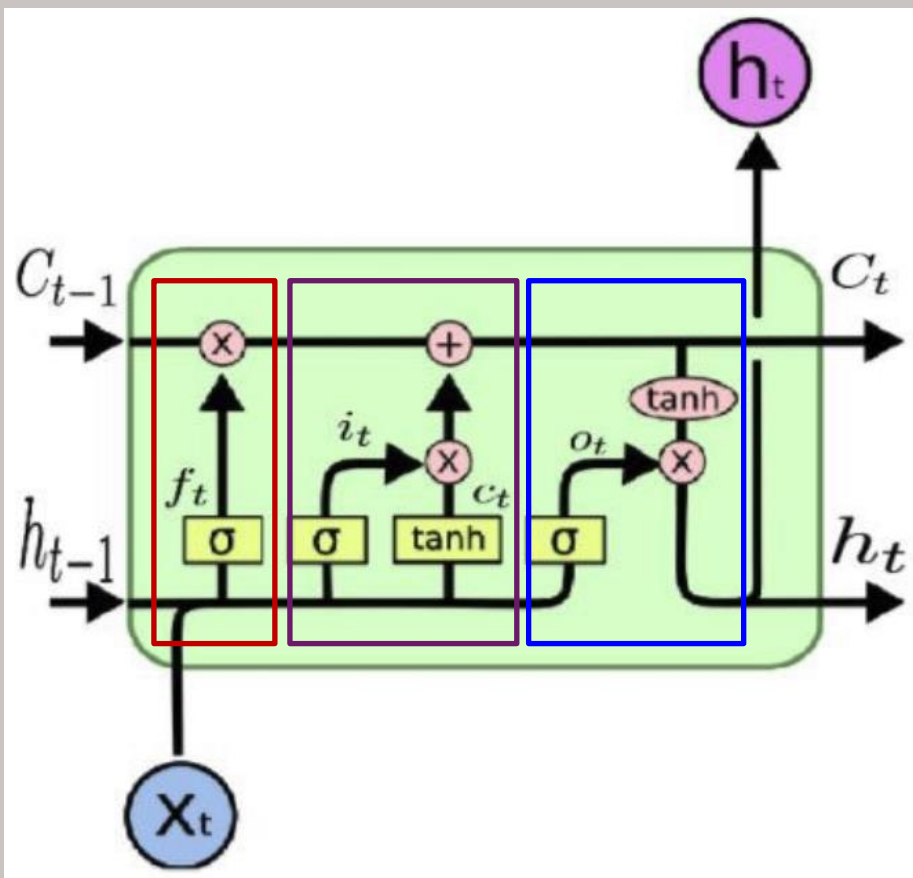
An LSTM has three of these gates, to protect and control the cell state.

Long Short-Term Memory (LSTM)



A simple LSTM cell consists of four gates:

- **Forget gate (f)** – whether and to what extent to forget (erase) the previous C_{t-1} cell
- **Input gate (i)** – it controls writing to the cell and how strong the given input influence the output result and combines it with the previous cell output
- **Output gate (o)** – how much to reveal the cell and use for computing the output h_t



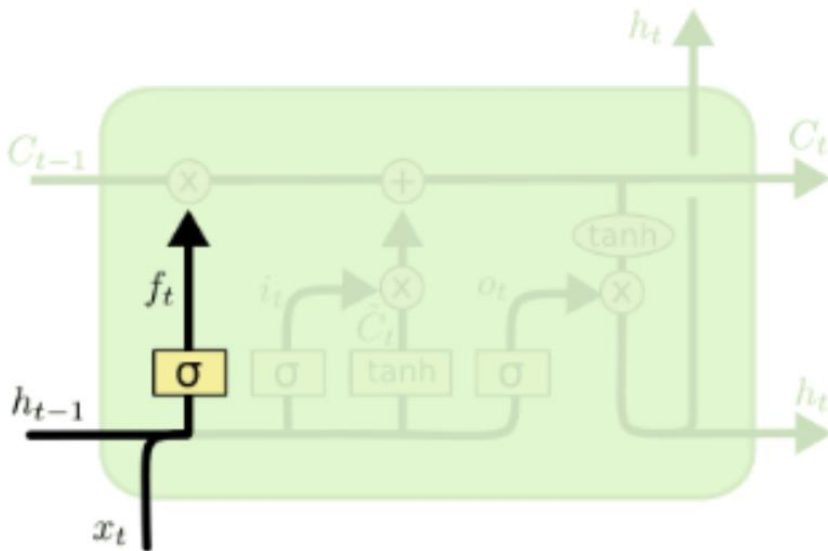
Long Short-Term Memory (LSTM)



In the first step, the LSTM decides by a sigmoid layer called the “forget gate layer” what information is let to go throw away from the cell state.

The **forget gate (o)** of a simple LSTM cell takes the decision about what must be removed from the h_{t-1} state after getting the output of the previous state, and it thus keeps only the relevant stuff. It is surrounded by a sigmoid function σ which crushes the input between $[0, 1]$.

We multiply the forget gate with previous cell state to forget the unnecessary stuff from the previous state which is not needed anymore.



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

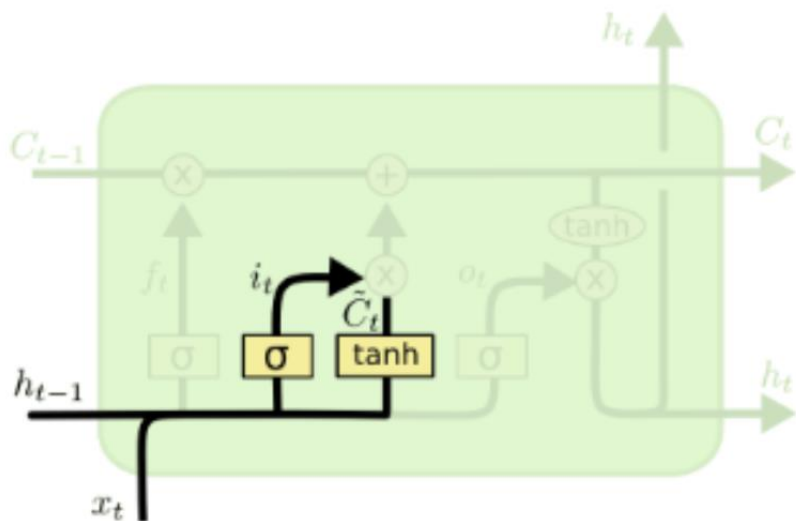
Long Short-Term Memory (LSTM)



In the next step, the LSTM decides what new information will be stored in the cell state: First, a **sigmoid layer** σ called the **input gate layer** decides which values we shall update. Next, a **tanh layer** creates a vector of new candidate values, \tilde{C}_t , that could be added to the state. In the next step, we shall combine these two to create an update to the state.

The **input gate (i)** of a simple LSTM decides about the addition of new stuff from the present input to our present cell state scaled by how much we wish to add them.

The sigmoid layer σ decides which values to be updated and **tanh** layer creates a vector for new candidates to added to the present cell state.



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

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

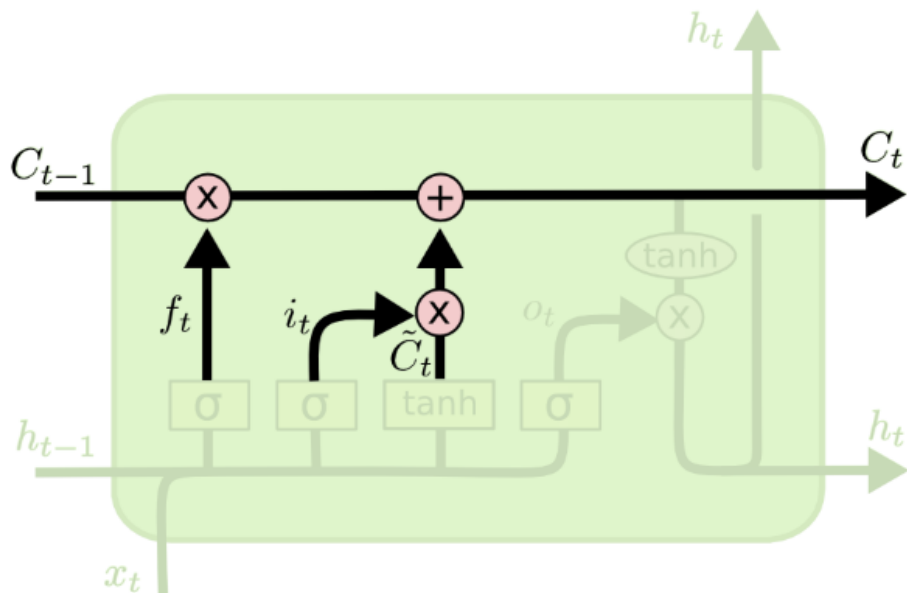
Long Short-Term Memory (LSTM)



In the third step, the LSTM updates the old cell state C_{t-1} into the new cell state C_t . The previous steps already decided what to do, we just need to actually do it.

We multiply the old state by f_t , forgetting the things we decided to forget earlier. Then we add $i_t * C_t$. This is the new candidate values, scaled by how much we decided to update each state value.

We can actually drop the information about the old subject's attribute and add the new information, as we decided in the previous steps.



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

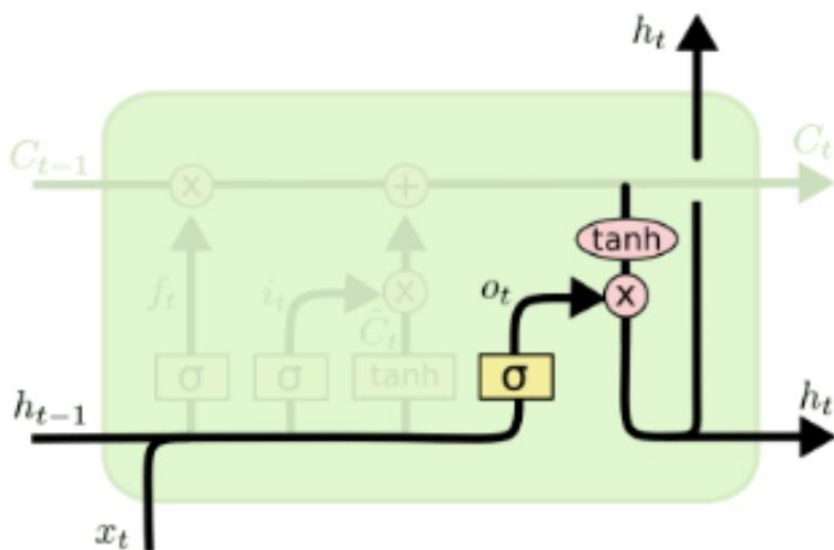
Long Short-Term Memory (LSTM)



Finally, the LSTM decides what is going to the output based on our cell state, but will be a filtered version. First, a sigmoid layer σ decides what parts of the cell state go to the output. Then, the cell state is put through \tanh (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that only the parts are sent to the output.

The **output gate (o)** of a simple LSTM cell decides what to output from the cell state which will be done by the sigmoid function σ .

The input x_t is multiplied with \tanh to crush the values between $(-1,1)$ and then multiply it with the output of sigmoid function:



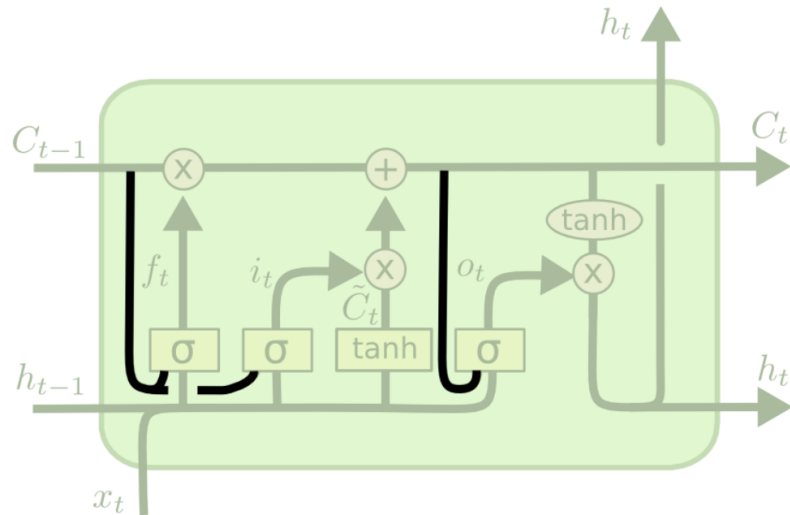
$$o_t = \sigma (W_o [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh (C_t)$$

Variants of LSTM



Peephole connections can be added to some or all the gates of the LSTM cells:

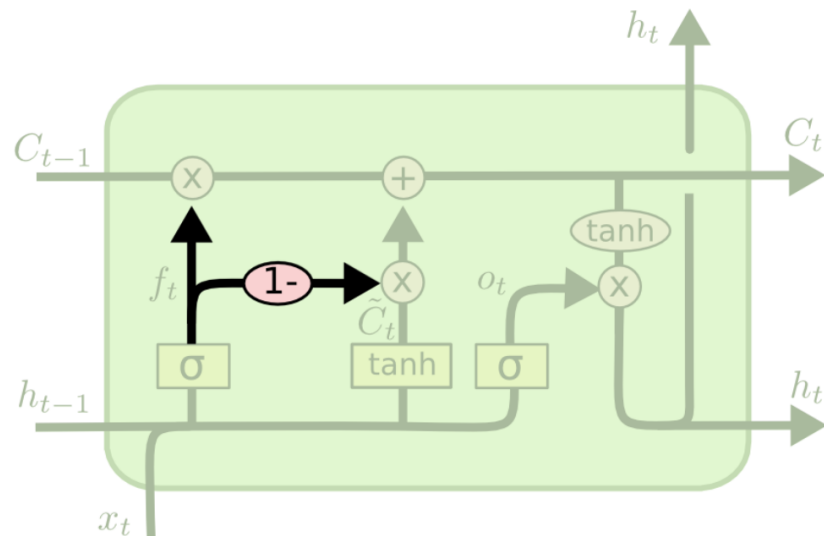


$$f_t = \sigma(W_f \cdot [C_{t-1}, h_{t-1}, x_t] + b_f)$$

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

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

The forget gate can be coupled to forget only when we are going to put something in the place of the forgotten older state:

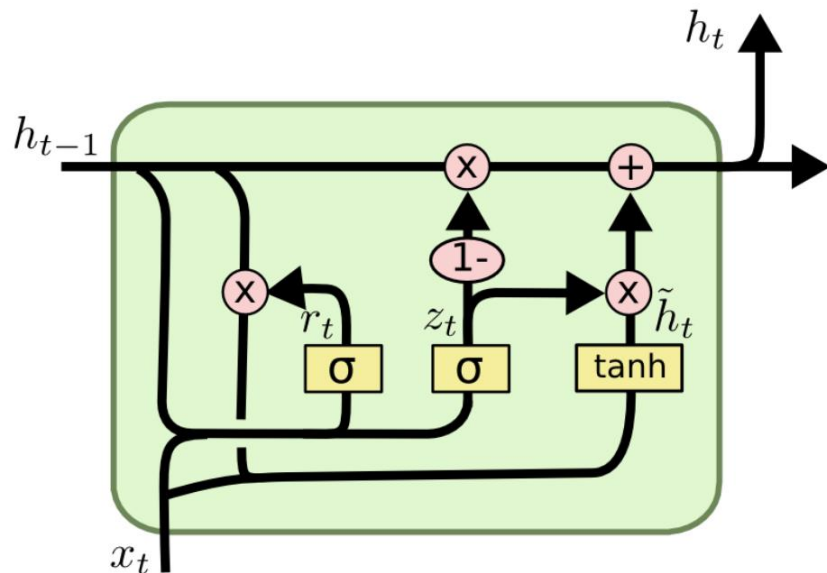


$$C_t = f_t * C_{t-1} + (1 - f_t) * \tilde{C}_t$$

Gated Recurrent Unit (GRU)



The gated recurrent unit combines the forget and input gates into a single update gate and merges the cell state and hidden state together with some other minor changes. In result the GRU units are simpler than LSTM ones:



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

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

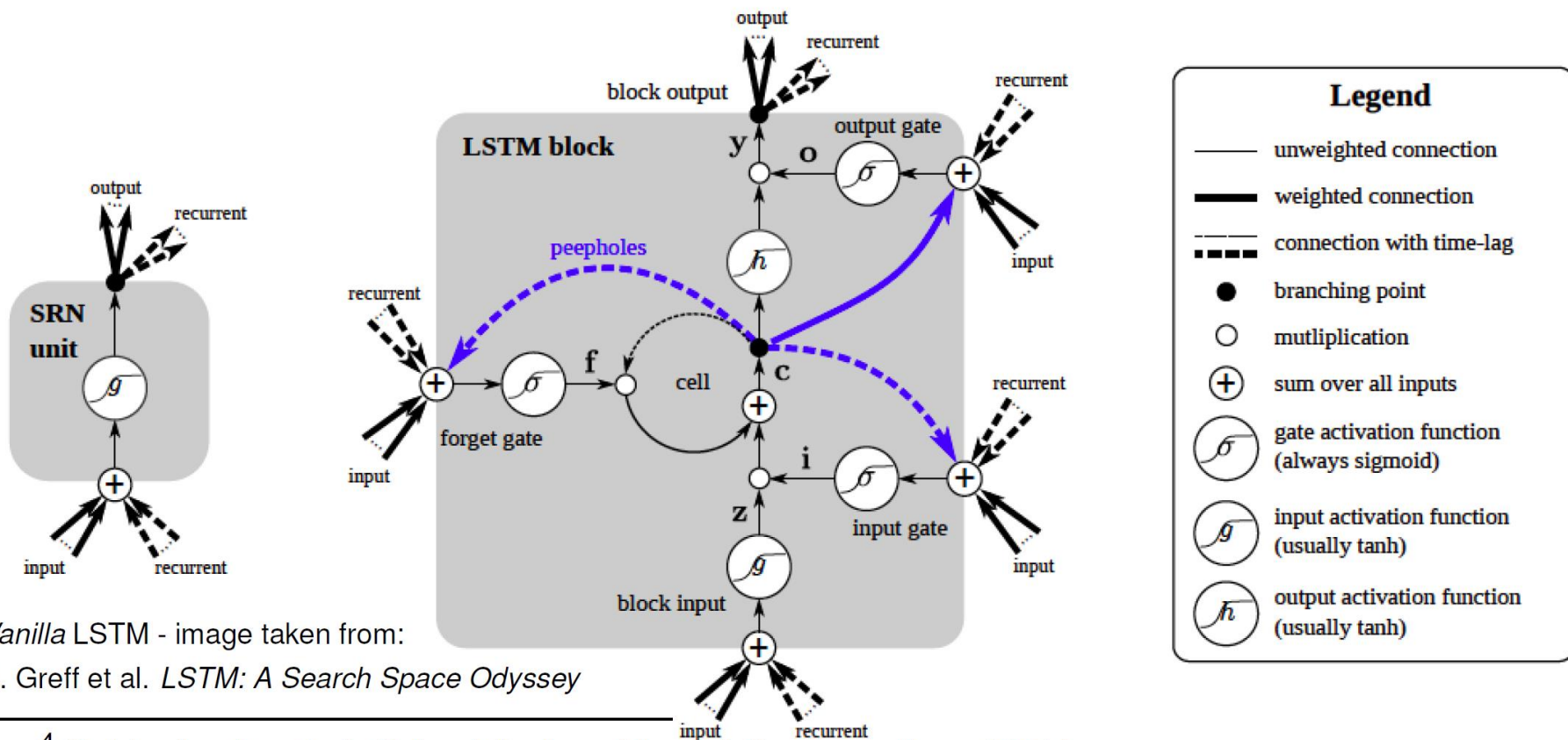
$$\tilde{h}_t = \tanh (W \cdot [r_t * h_{t-1}, x_t])$$

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

Long Short-Term Memory (LSTM)



LSTM is an extension of RNN that can deal with long-term temporal dependencies. It implements a mechanism that allows the networks to “remember” relevant information for a long period of time:



Vanilla LSTM - image taken from:
K. Greff et al. *LSTM: A Search Space Odyssey*

⁴ S. Hochreiter & J. Schmidhuber, *Neural Computation*, 1997

Long Short-Term Memory (Vanilla LSTM)



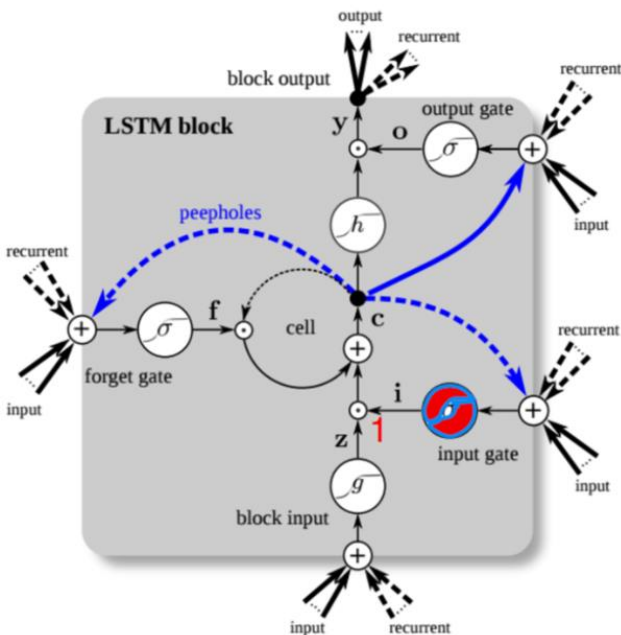
Exploits a linear memory cell (state) that integrates input information through time:

- memory obtained by self-loop,
- gradient not down-sized by Jacobian of sigmoidal function → no vanishing gradient!

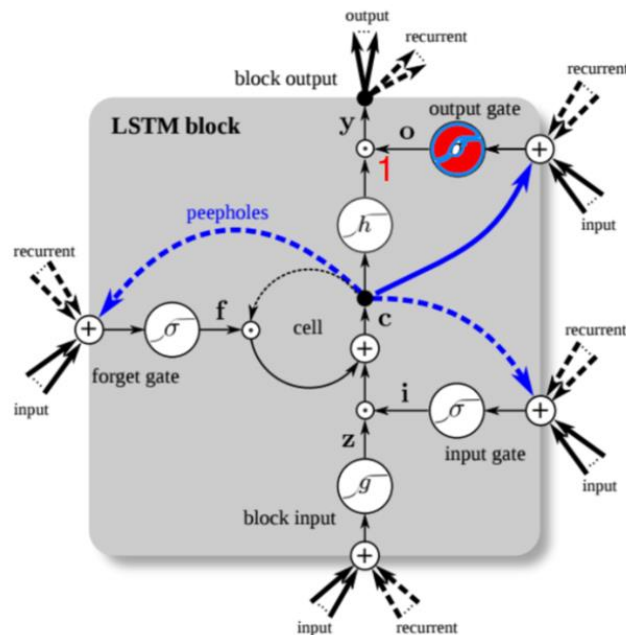
3 gate units with sigmoid soft-switch control the information flow via multiplicative connections:

- input gate “on”: let input to flow in the memory cell,
- output gate “on”: let the current value stored in the memory cell to be read in output,
- forget gate “off”: let the current value stored in the memory cell to be reset to 0.

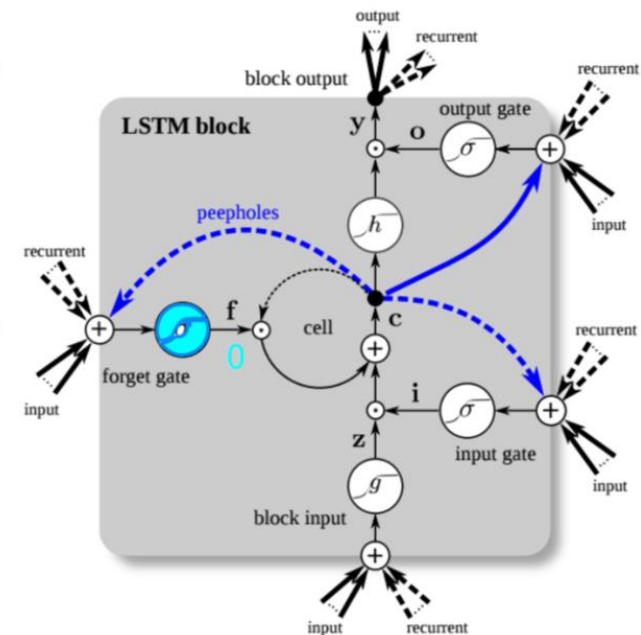
input gate on



output gate on



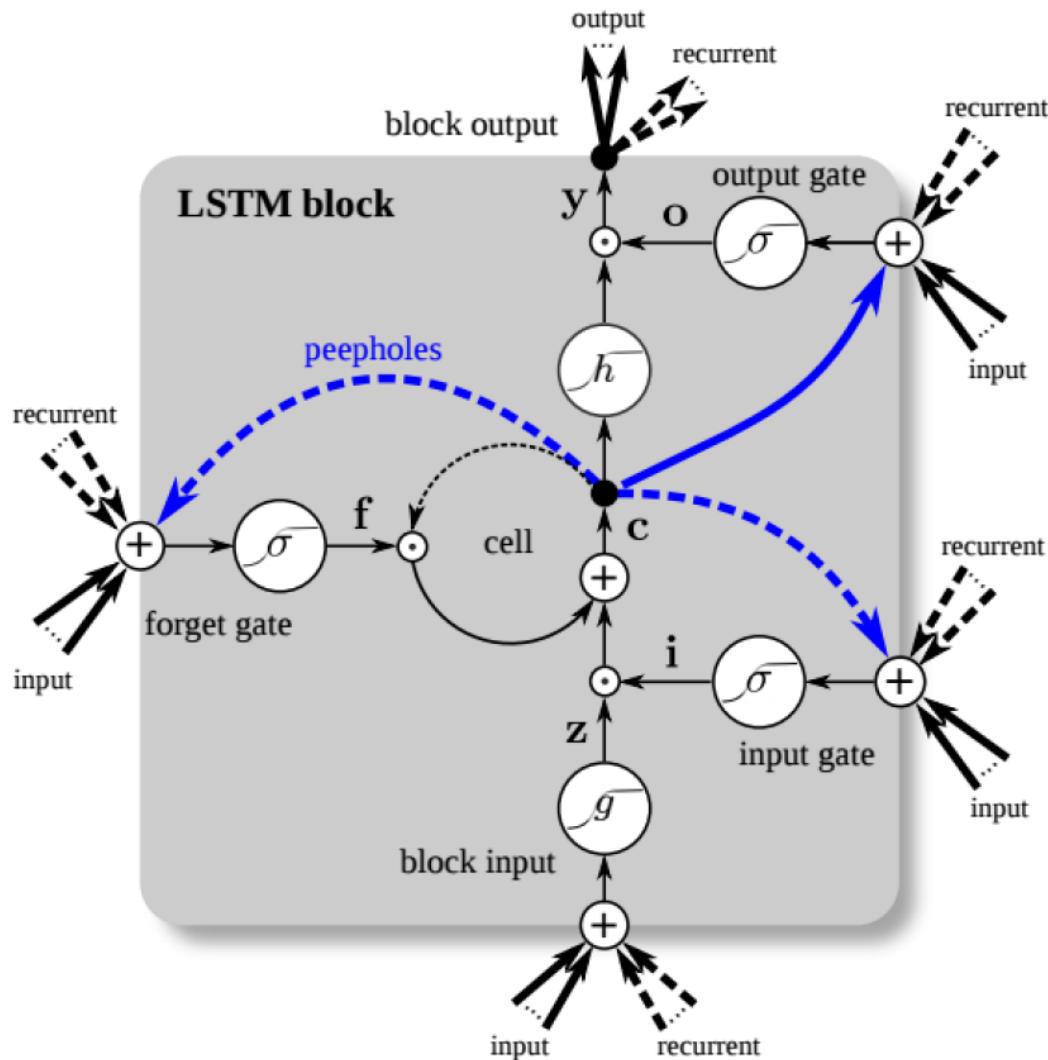
forget gate off



Long Short-Term Memory (Vanilla LSTM)



Peepholes connections of Vanilla LSTM allow directly controlling all gates to easier learn precise timings, supporting full backpropagation through time training:



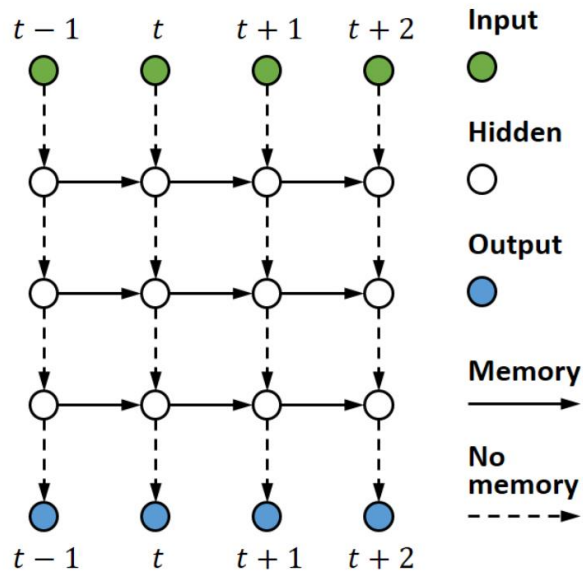
Legend

- unweighted connection
- weighted connection
- - - connection with time-lag
- branching point
- ⊙ multiplication
- ⊕ sum over all inputs
- ⊗ gate activation function (always sigmoid)
- ⊗ input activation function (usually tanh)
- ⊗ output activation function (usually tanh)

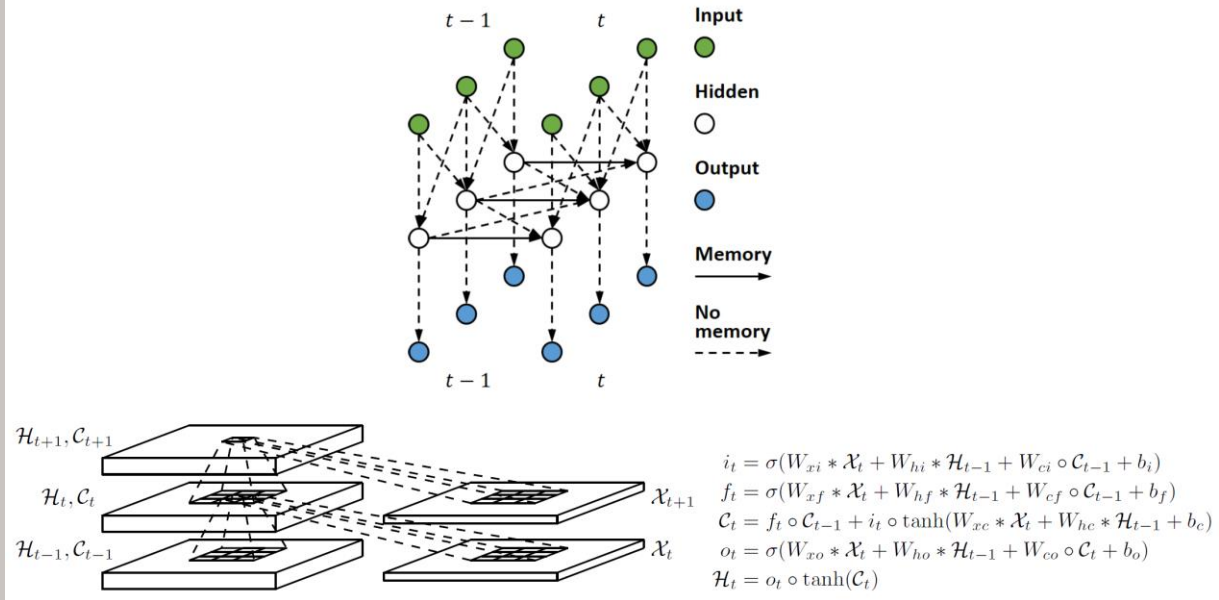
Deep Grid and Convolutional LSTM Networks



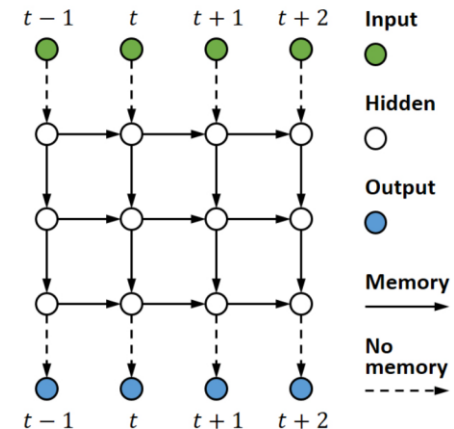
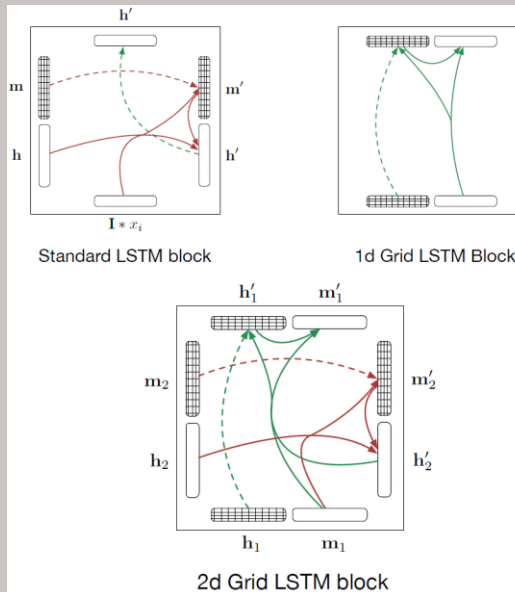
Stacked LSTM (sLSTM)



Convolutional LSTM (cLSTM)



Grid LSTM where cells are connected between network layers as well as along the spatiotemporal dimensions of the data:





Bibliography and Literature

1. Ian Goodfellow, Yoshua Bengio, Aaron Courville, **Deep Learning**, MIT Press, 2016, ISBN 978-1-59327-741-3 or PWN 2018.
2. Holk Cruse, [Neural Networks as Cybernetic Systems](#), 2nd and revised edition
3. R. Rojas, [Neural Networks](#), Springer-Verlag, Berlin, 1996.
4. [Convolutional Neural Network](#) (Stanford)
5. [Visualizing and Understanding Convolutional Networks](#), Zeiler, Fergus, ECCV 2014
6. [Lectures of Alessandro Sperduti](#) of Universita Degli Studi di Padova
7. [Exploding Gradient Problem](#)
8. [LSTM cells from scratch and the code](#)
9. [Understanding LSTM](#)



Adrian Horzyk

horzyk@agh.edu.pl

Google: [Horzyk](#)



**University of Science
and Technology
in Krakow, Poland**

