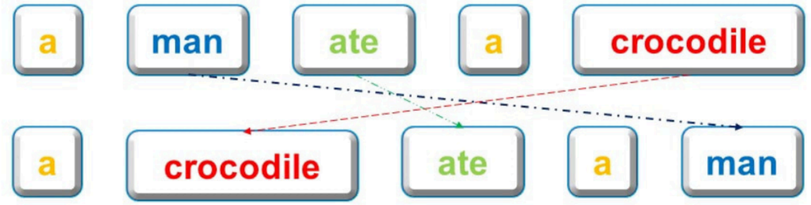
Lecture 6: Networks for Sequential Modeling

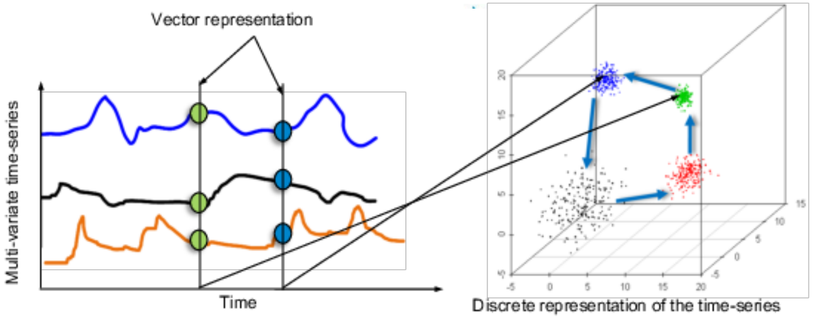
Program for Today

Date	Topic	Tutorial
Sep 17	Intro & class description	Linear Algebra in a nutshell + prob and stat
Sep 24	Basic of machine learning + Dense NN	*Basic jupyter + DNN on mnist (give jet dnn as homework)*
Oct 1	Convolustional NN	*Convolutional NNs with MNIST*
Oct 8	Training in practice: regularization, optimization, etc	*Practical methodology*
Oct 15	*Google tutorial*	
Oct 22	Recurrent NN	*Hands-on exercise*
Oct 29	Graph NNs	*Tutorial on Graph NNs*
Nov 5	Unsupervised learning and anomaly detection	*Autoencoders with MNIST*
Nov 12	Generative models: GANs, VAEs, etc	Normalizing flows
Nov 19	Transformers	
Nov 26	Network compression (pruning, quantization, Knoweldge Distillation)	
Dec 3	Tutorial on hls4ml/qkeras	
Dec 10	Quantum Machine Learning	
Dec 17	Q/A Prior to exam	

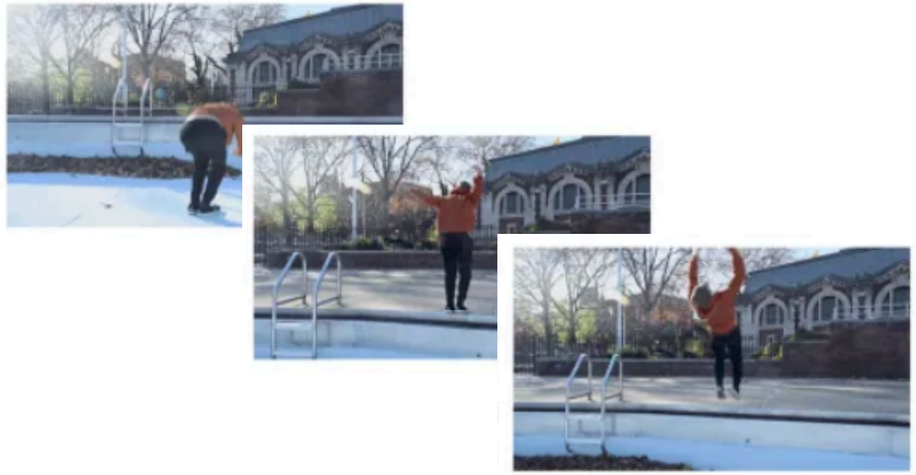
Sequential Data

- *Sequential data are arrays of ordered values, in which there is an intrinsic metric that dictates what comes before what*
  - *Words ordered in a sentence according to grammar*
  - *Sensor values sampled at fixed time intervals*
  - *Genetic sequences*
  - *Movies (a sequence of image frames)*
  - *Sound*
  - *…*
- *These data are represented as ordered array* $x^{(1)}, \ldots, x^{(\tau)}$
  - *Each instance* $x^{(i)}$ *in the sequence is itself a vector of features*
    - *Similar to channels in images*
    - *Example, the letters of a word or multiple readout values from a set of sensors*



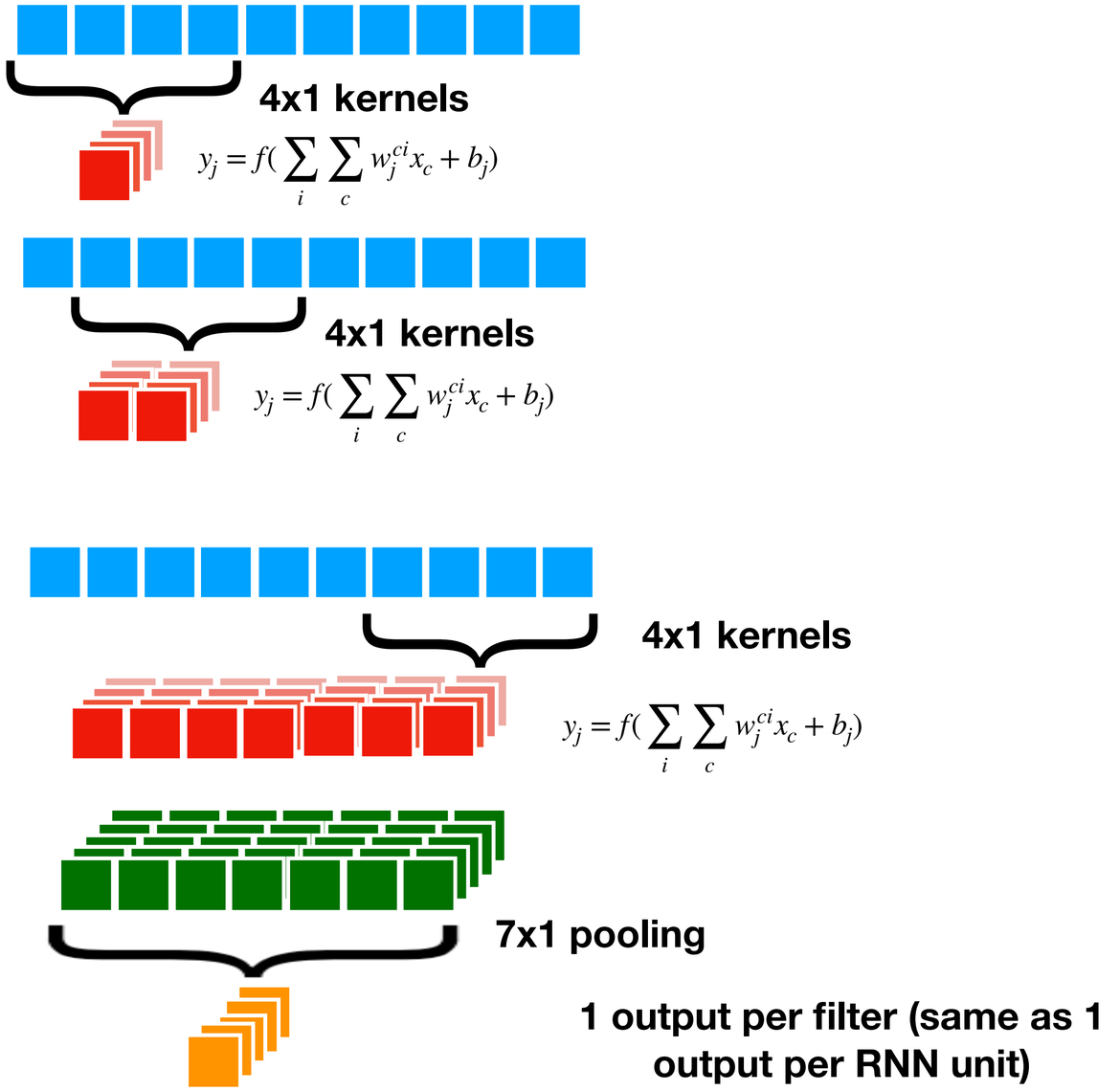


A C A A G C C T G T C C A A C T



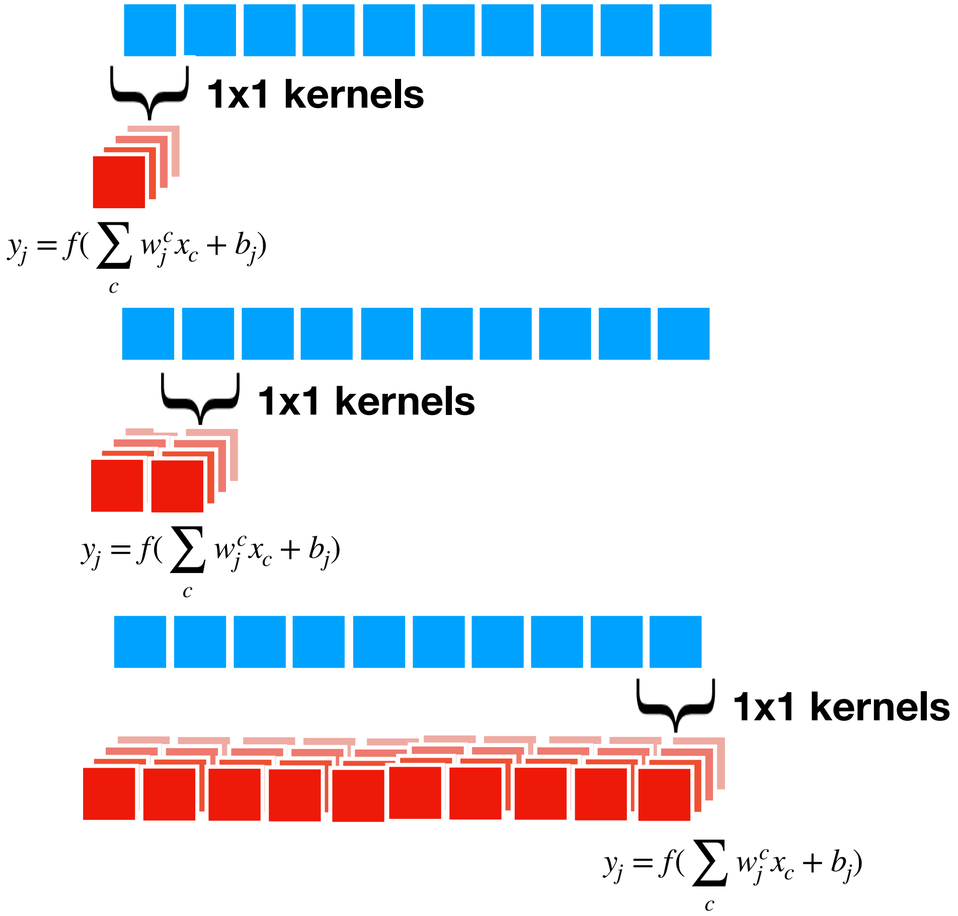
Sequential Data with CNNs

- One could process sequential data with a 1D kernel
  - Same as 2D conv, but acting on 1D sequence
  - The channels here are the features of the i-th element of the sequence (e.g., multiple sensor readouts)
  - The same concept applies
    - stride, pooling, padding, etc.



1D kernels as feature extractors

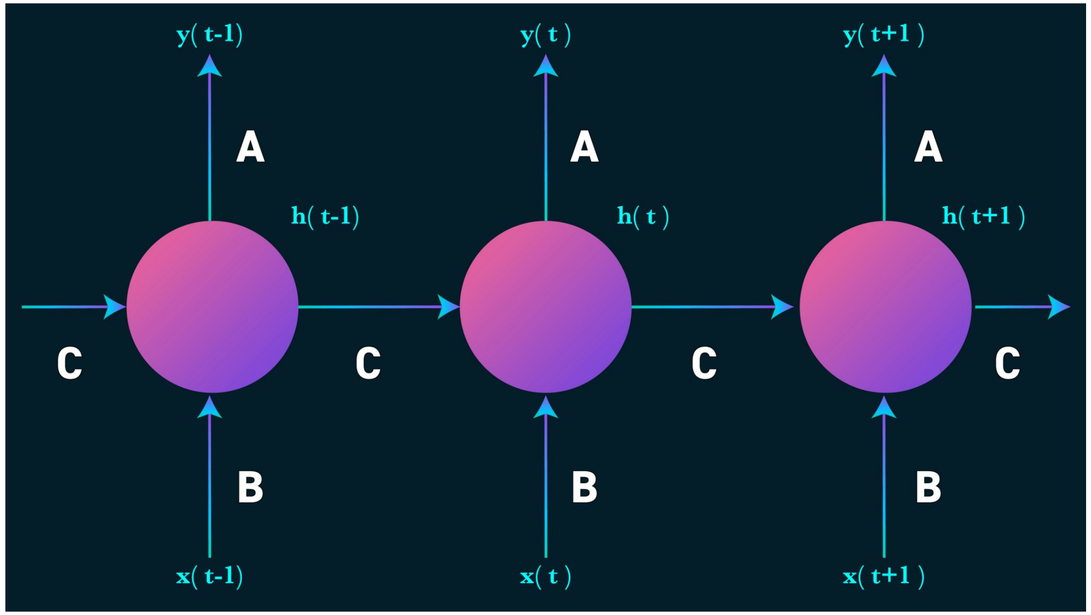
- *Sometimes, 1x1 kernels are used (in 1D, 2D, etc) as pre-processing networks*
  - *If you look at the math, it's the same as running a DNN in parallel across the various channels for that specific pixel*
  - *The same network processes all of them*
  - *This is used to change the basis of inputs from a vector x to a vector $\tilde{x} = f(x)$*



Recurrent Neural Networks

- While other architectures could process these data (DNNs, 1D CNNs,…) RNNs do so with special mechanisms, designed to play the role of memory across the sequence
  - They are then particularly suitable for long sequences
- The essence aspect is parameters sharing (that we already discussed for CNNs)
  - It allows to retrieve information regardless of the position in the sequence (like translation equivariance in CNNs, and unlike DNNs)
  - When can process examples of variable length (e.g., a sentence) with the same network (not true for DNNs and CNNs)

https://towardsdatascience.com/understanding-rnn-and-lstm-f7cdf6dfc14e



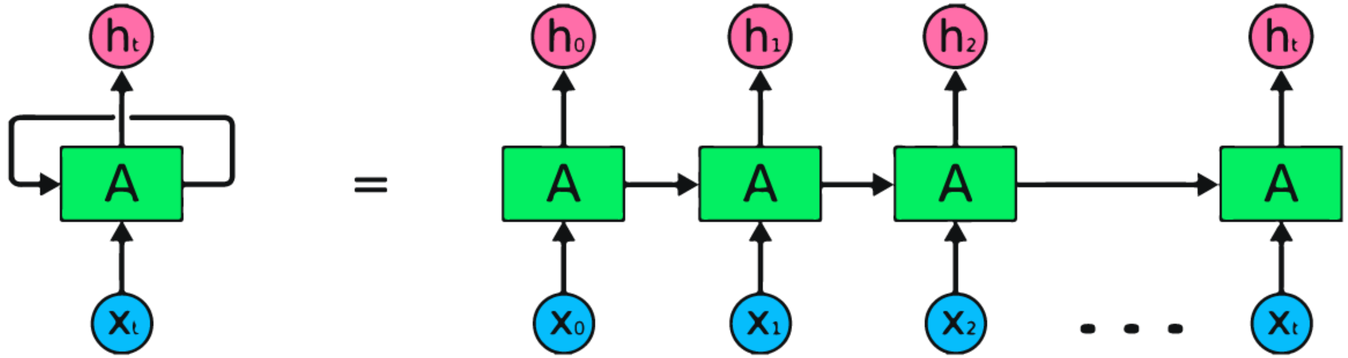
I went to Nepal in 2009

In 2009, I went to Nepal

A comparison

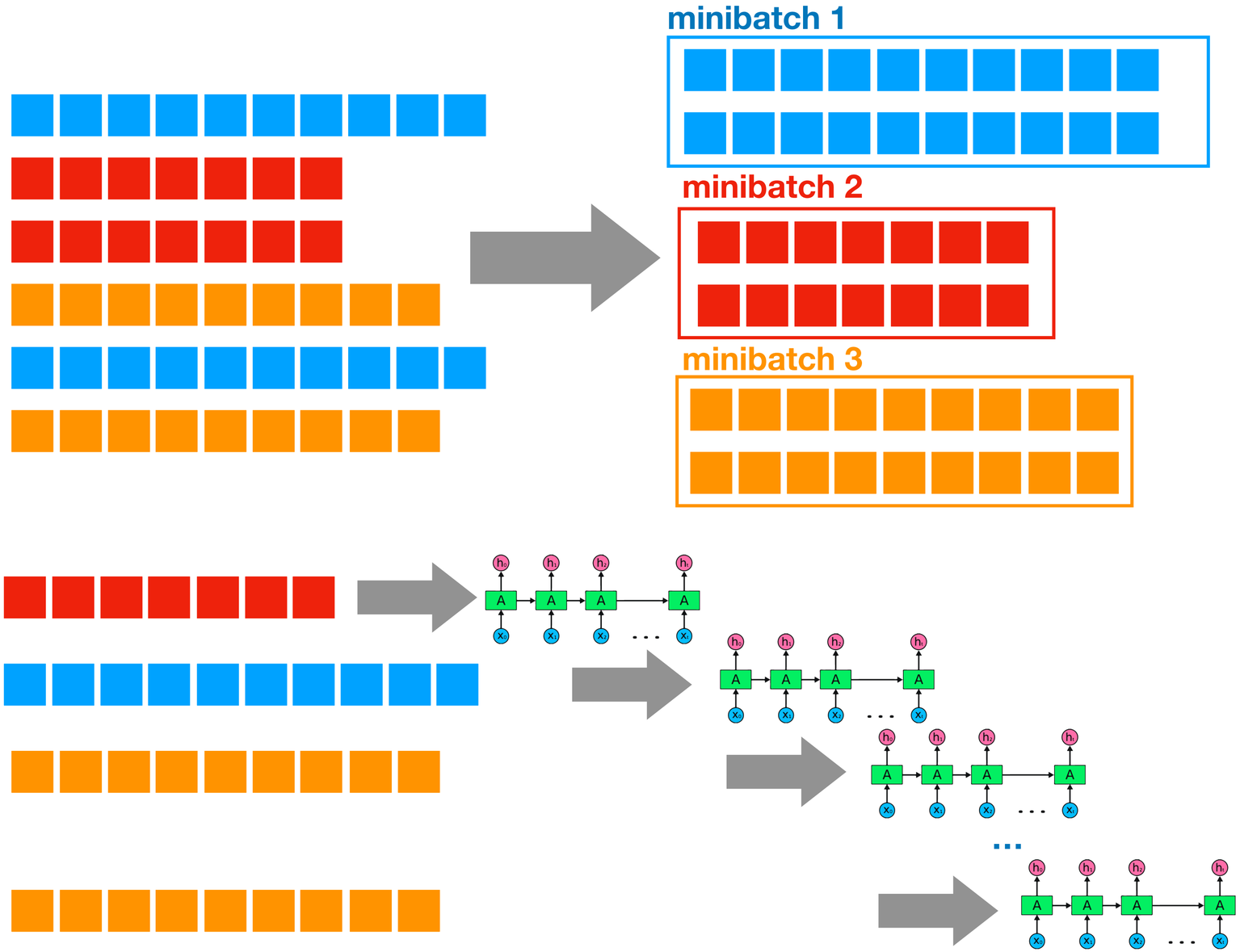
- ***1D CNNs***
  - *process the data in parallel (faster)*
  - *share parameters (the same "DNN-like" processing to all pixels) locally: each output pixel as a function of a few near neighbours (depending on the kernel size)*
  - *The sequence has to include examples of fixed length*

- ***Recurrent networks***
  - *process data sequentially (slower) since the process of element i-th depends on the output of element i-1*
  - *share parameters across the entire sequence, so that each output i-th is a function of all the elements j<i:* $y^{(\tau)} = f(f(f(\ldots f(x^{(i)})\ldots)))$
    - *Parameters are shared through a deeper computational graph*
  - *The sequence can contain examples of various length*



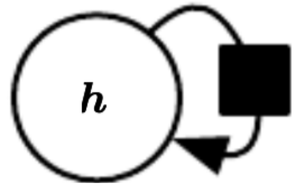
When using RNN in practice

- *At Training time*
  - *Use Minibatch*
  - *Each minibatch should be of fixed length*
- *At inference*
  - *use N=1 minibatches, so that the sequence duration can be arbitrary*
  - *Inference will be slower*
    - *Serialised across examples*
    - *Serialised across elements within an example*

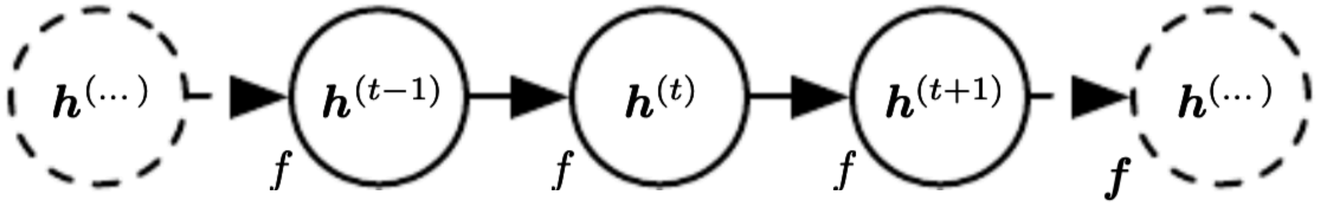


Computational Graphs

- Consider a structure of concatenated operations
  - e.g., a dynamic system whose next state is a function of the current state: $h^{(t)} = f(h^{(t-1)}|\theta)$

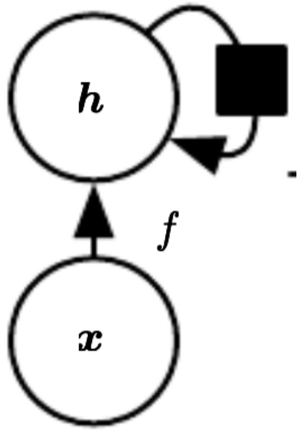


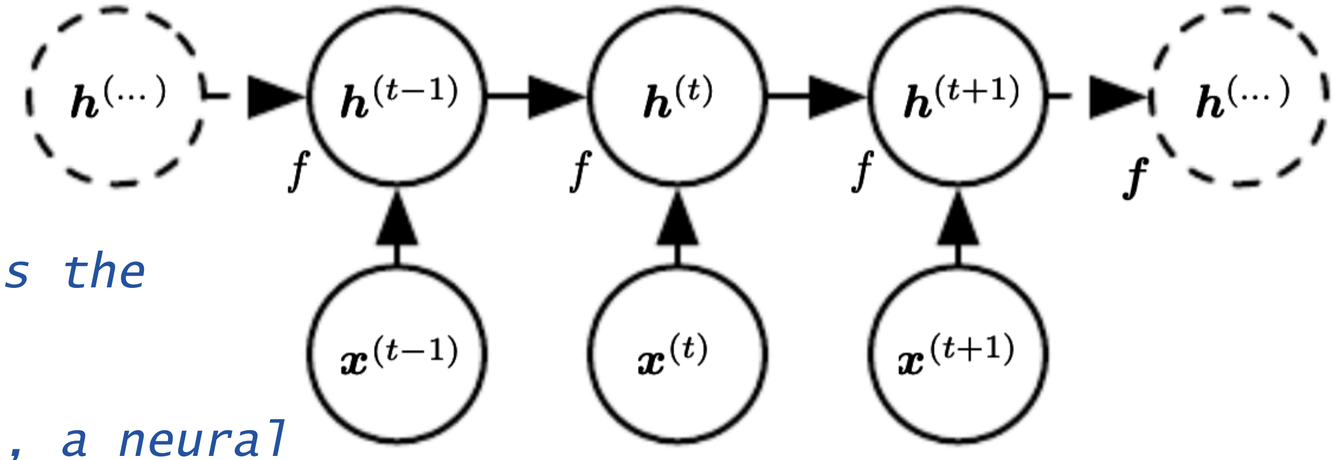
- An n-step computation can be unfolded in a graph of n single-step computations
  - $h^{(1)} = f(h^{(0)}|\theta)$
  - $h^{(2)} = f(h^{(1)}|\theta) = f(f(h^{(0)}|\theta)|\theta)$
  - ...



Add an external input

- Consider a structure of concatenated operations, whose output depends on some external input
  - In this case $h^{(t)} = f(h^{(t-1)}, x^{(t)} | \theta)$
- An n-step computation can be unfolded in a graph of n single-step computations
  - $h^{(1)} = f(s^{(0)}, x^{(1)} | \theta)$
  - $h^{(2)} = f(h^{(1)}, x^{(2)} | \theta) = f(f(h^{(0)}, x^{(1)} | \theta), x^{(2)} | \theta)$
  - …
- What defines the sequence processing is the function f
- It could be a learnable function, e.g., a neural network trained on data. This is what we call a RNN





Various kinds of RNNs

- *Several architectures proposed. What changes is what happens in the A block*
  - *SRNs, aka Elman (1990) and Jordan (1997) networks: simplest realisation of the idea*
  - *LSTMs (1995): the most popular (and performing) choice for serial-data processing*
  - *GRUs (2014): essentially an LSTM with a forget gate, with less parameters (and similar performance) as LSTM*

From Keras 2018

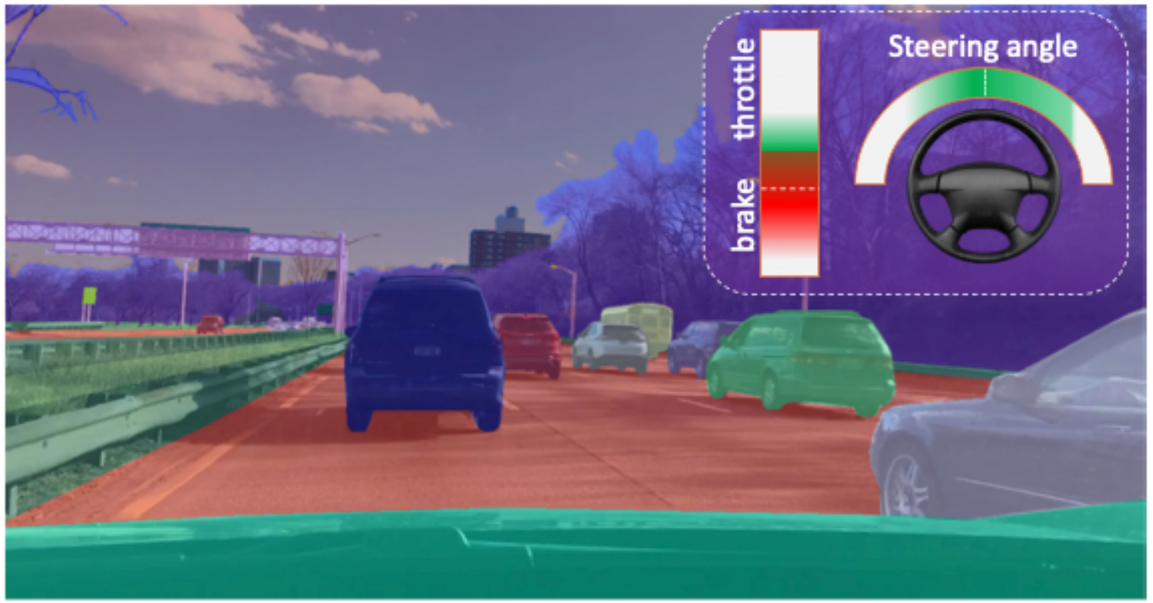
Recurrent layers

- LSTM layer
- GRU layer
- SimpleRNN layer
- TimeDistributed layer
- Bidirectional layer
- ConvLSTM2D layer
- Base RNN layer

From Keras 2024

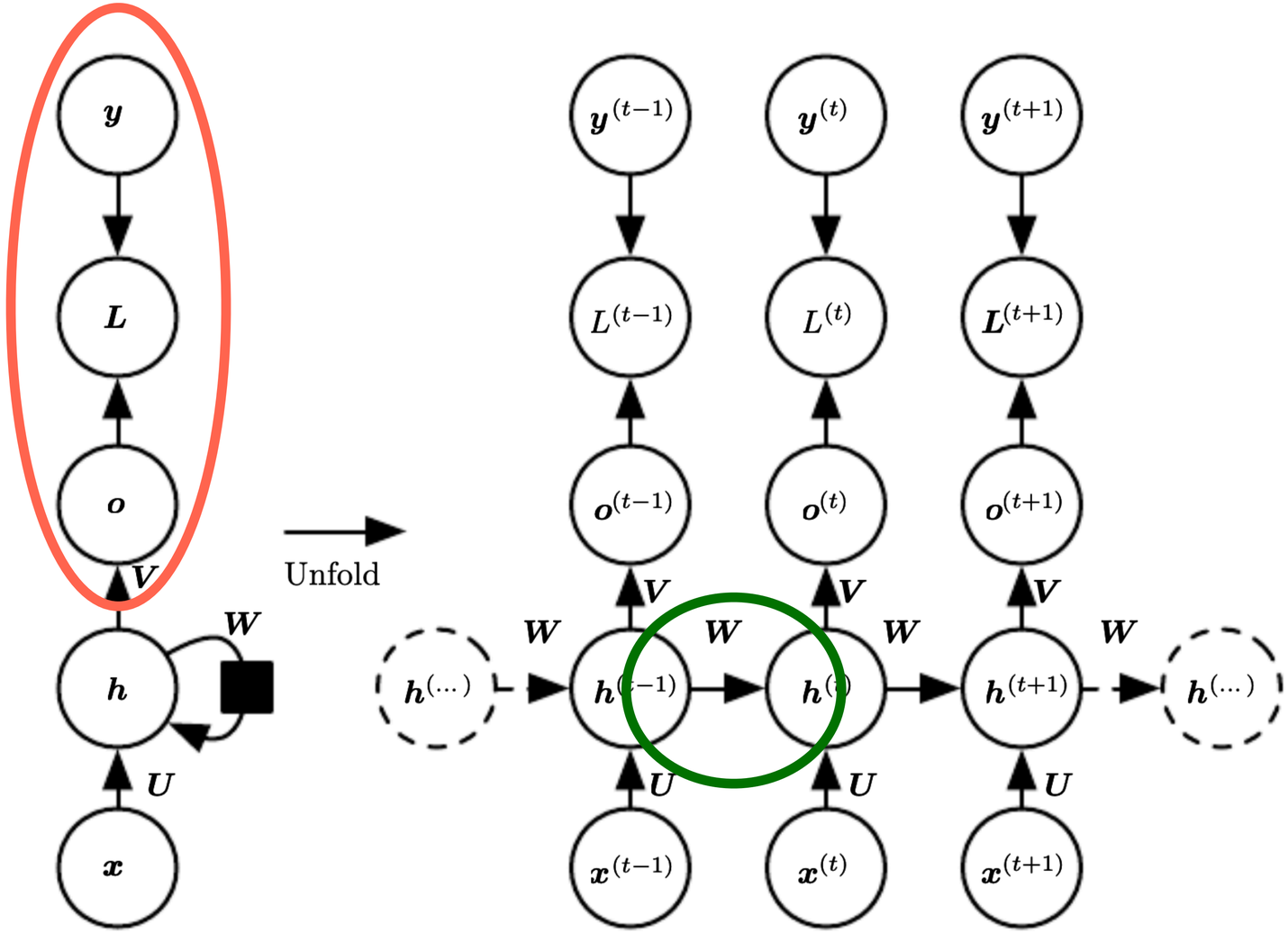
Recurrent layers

- LSTM layer
- LSTM cell layer
- GRU layer
- GRU Cell layer
- SimpleRNN layer
- TimeDistributed layer
- Bidirectional layer
- ConvLSTM1D layer
- ConvLSTM2D layer
- ConvLSTM3D layer
- Base RNN layer
- Simple RNN cell layer
- Stacked RNN cell layer



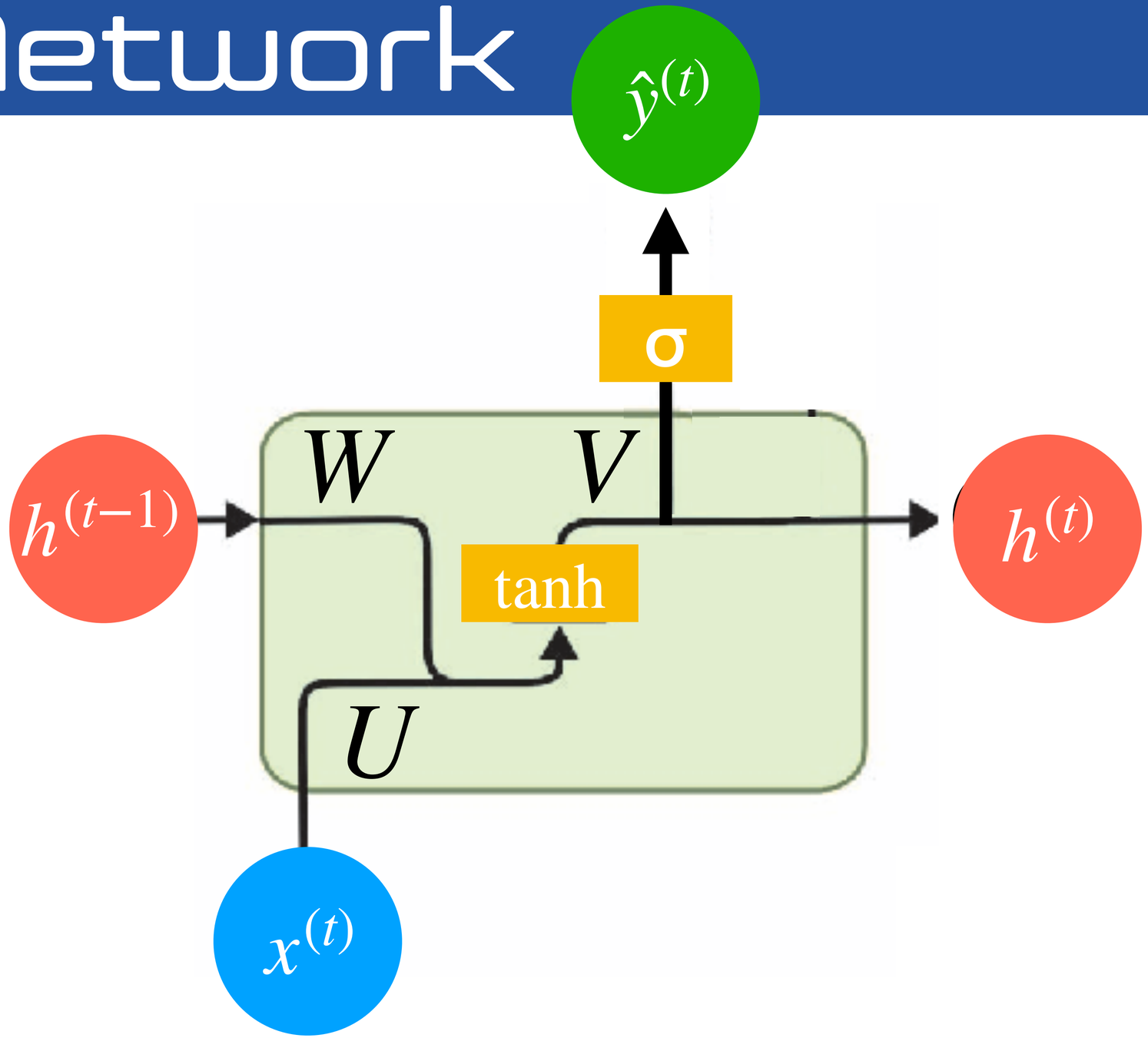
Elman Network

- One can have
  - an output per unit, which is compared to a label to compute a loss (e.g., at training time)
  - connections between hidden units
- One has to specify what goes in the computation unit



Elman Network

- Severable learnable parameters
  - biases b and c
  - Weight matrices
    - Input-to-hidden U
    - Hidden-to-output V
    - Hidden-to-hidden W



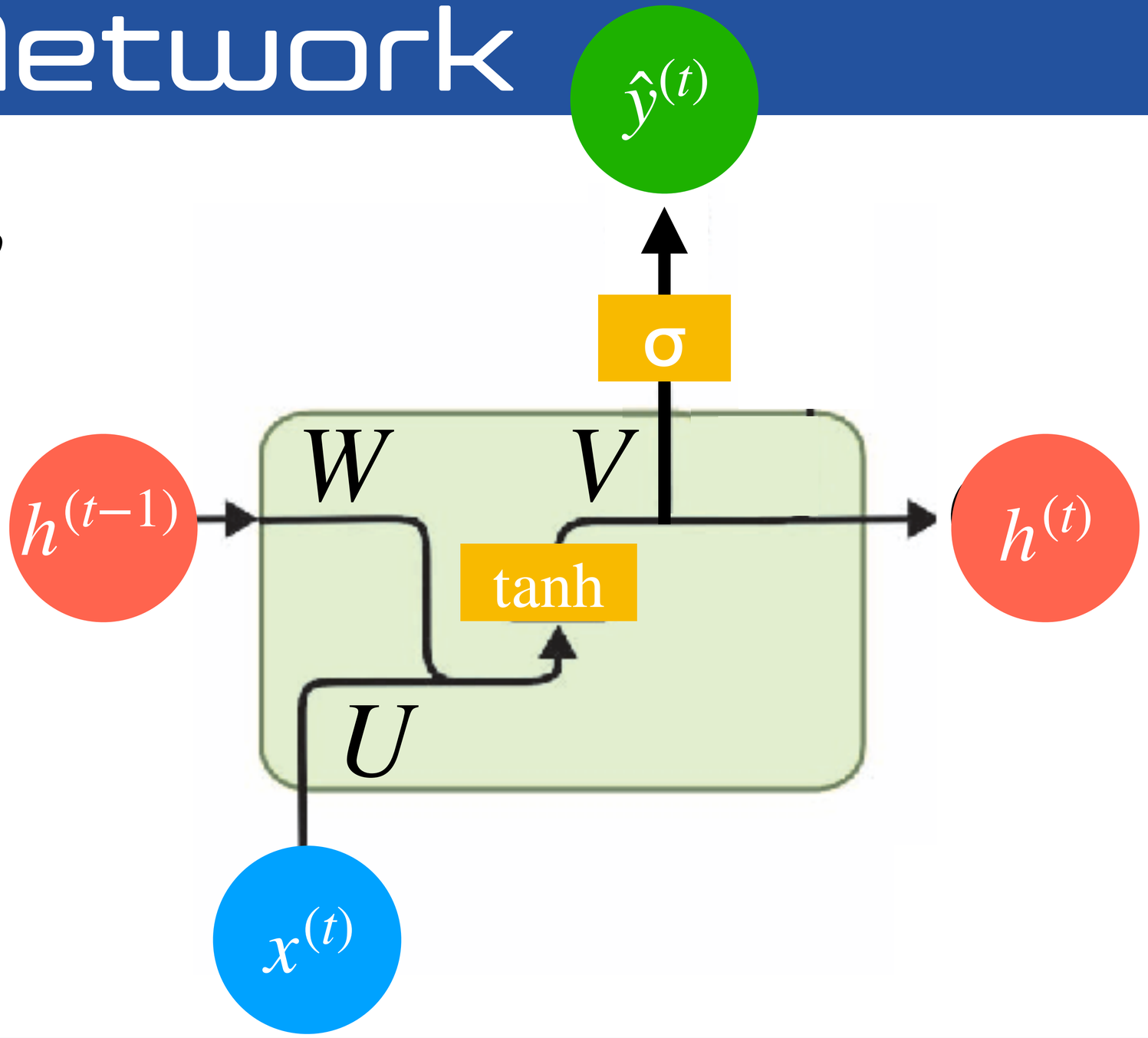
Elman Network

$$a^{(t)} = Wh^{(t-1)} + Ux^{(t)} + b$$

$$h^{(t)} = \tanh(a^{(t)})$$

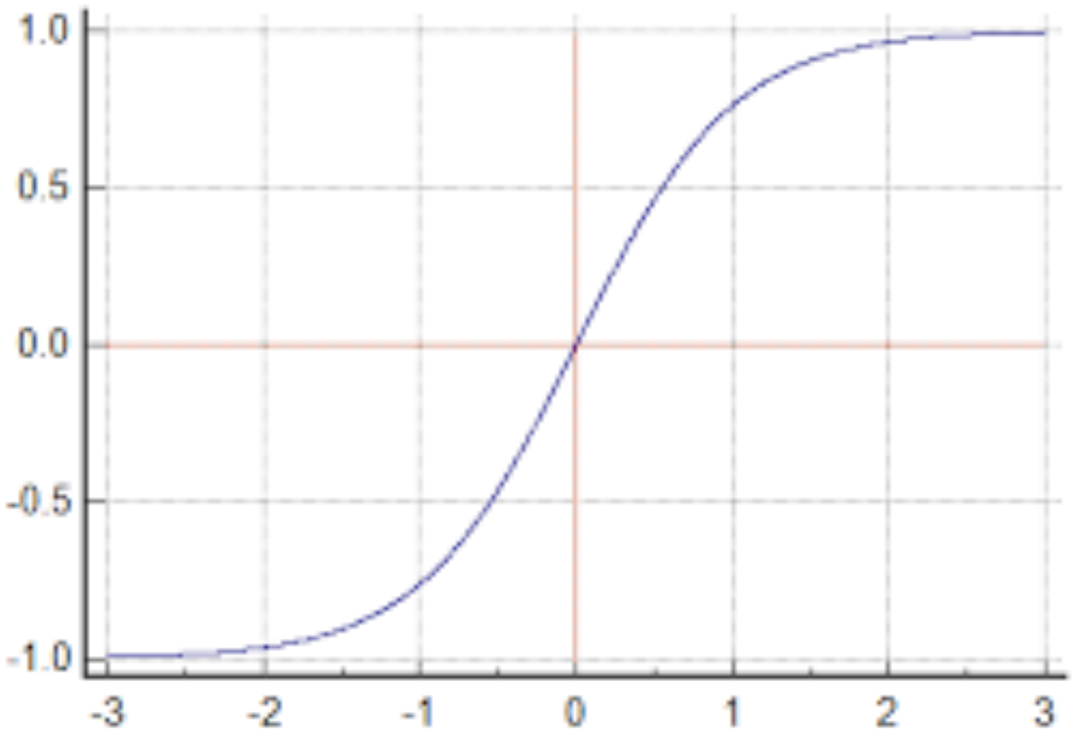
$$o^{(t)} = Vh^{(t)} + c$$

$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$



Why tanh and σ?

- *The network processing essentially works as memory gates*
- *One typically uses sigmoids or tanh as activation functions to have the gate functionality*
  - *What changes is the definition region for the output, [0, 1] vs [-1, 1]*



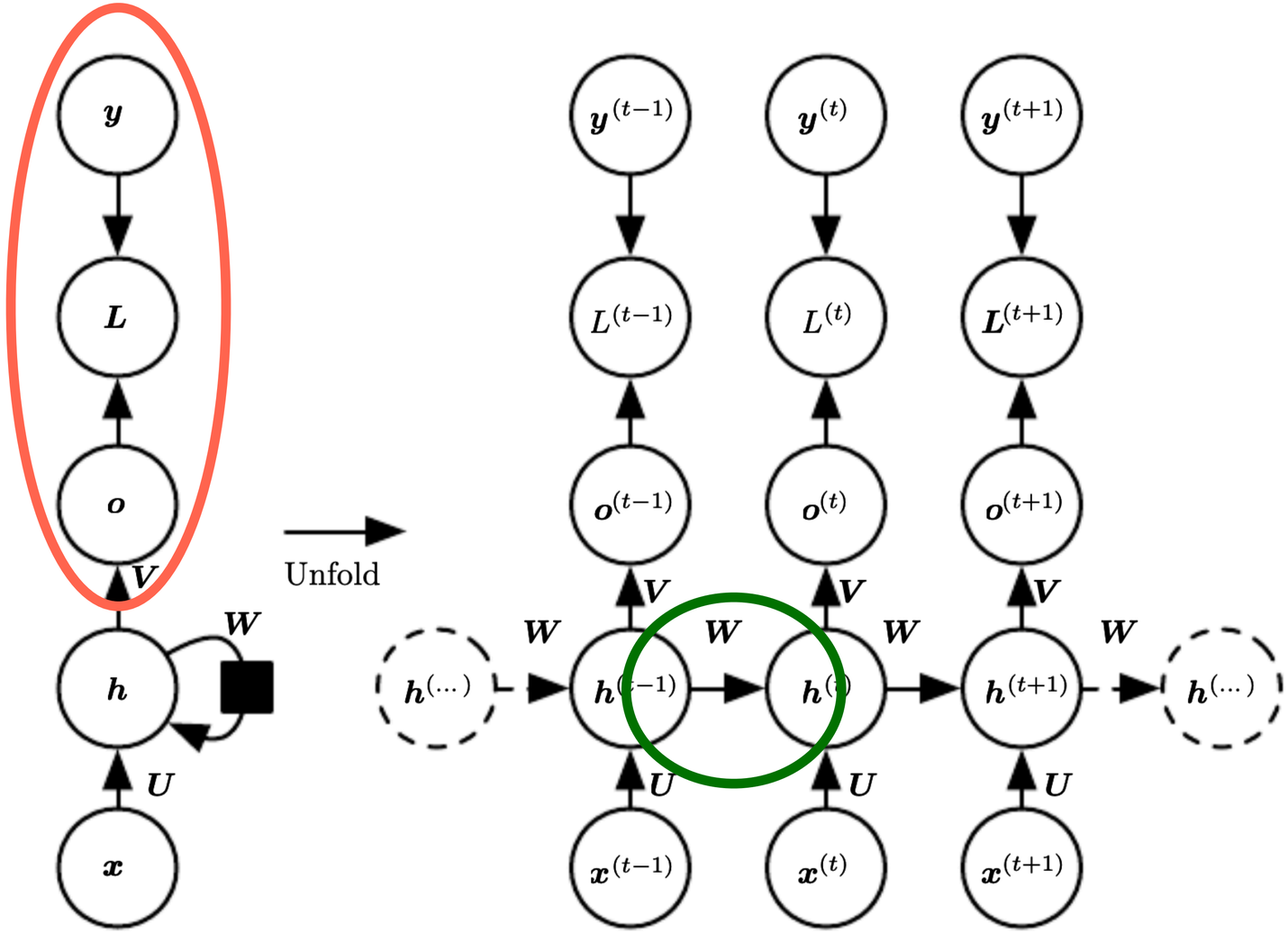
Loss and Gradient

- *Elman's network are used when one has some ground truth which is also a sequence*
  - *One compares the output at each time t*
  - *Apply the softmax to it*
  - *Compares it to the ground truth at same time t to compute the loss*
- *Gradient evaluation is expensive*
  - *Intrinsically sequential*
  - *Reaquires sto store intermediate values in memory*

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

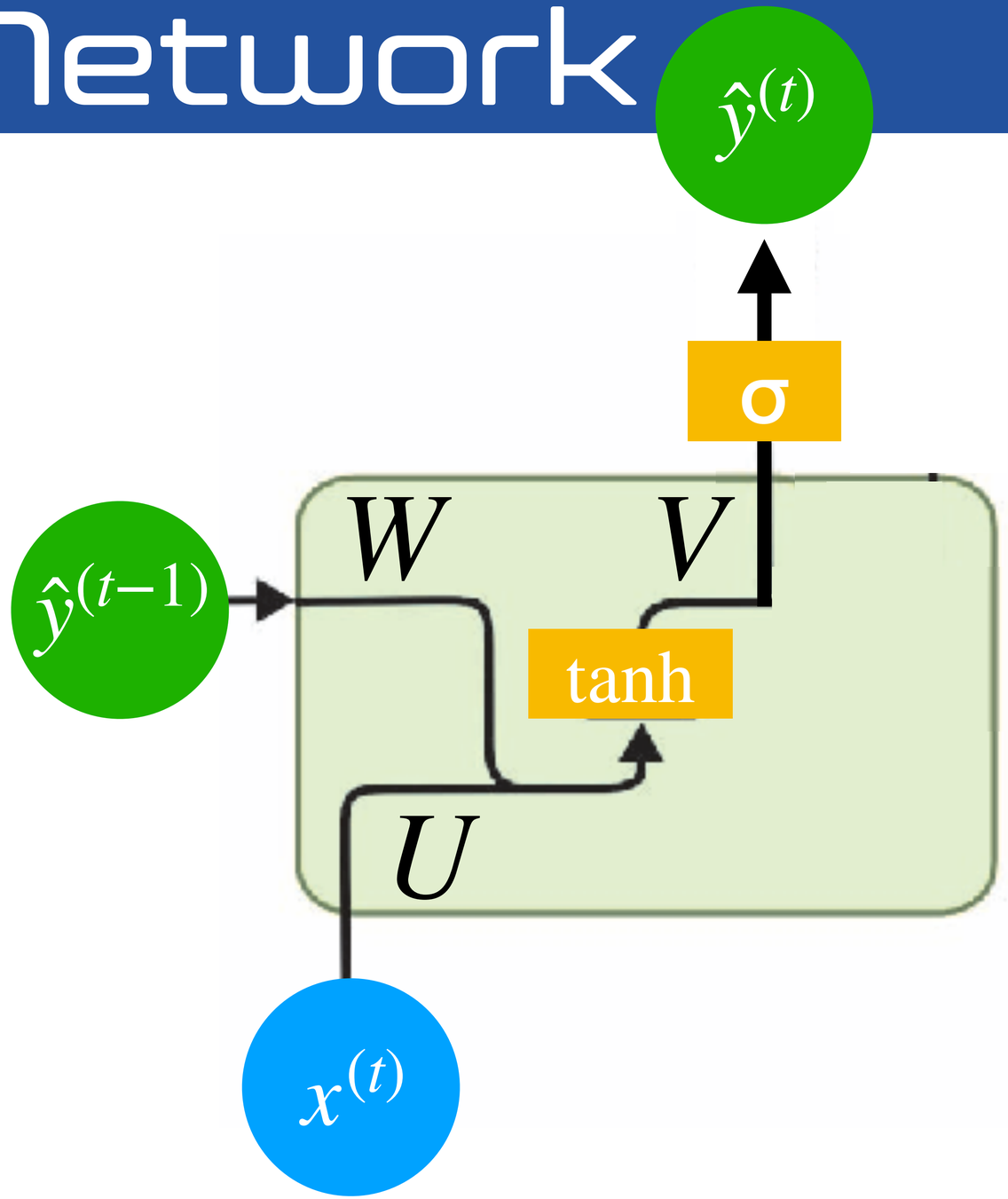
Jordan Network

- One can have
  - an output per unit, which is compared to a label to compute a loss (e.g., at training time)
  - connections between output and hidden units, i.e., no hidden-to-hidden
- Besides this difference, computation unit like for Edman's loss



Jordan Network

- Severable learnable parameters
  - biases b and c
  - Weight atrices
    - Input-to-hidden U
    - Hidden-to-output V
    - Hidden-to-hidden W



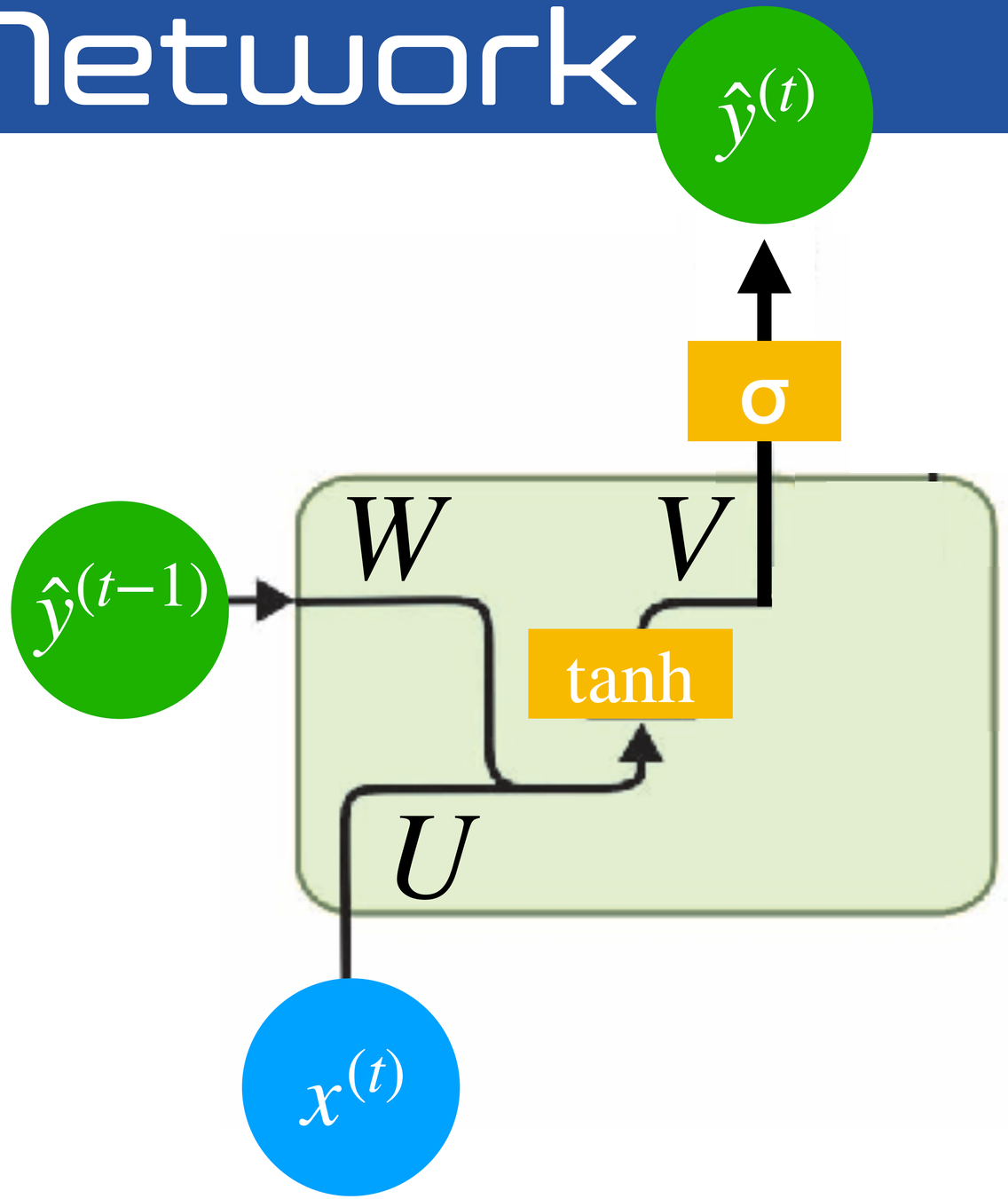
Jordan Network

$$a^{(t)} = Wh^{(t-1)} + Ux^{(t)} + b$$

$$h^{(t)} = \tanh(a^{(t)})$$

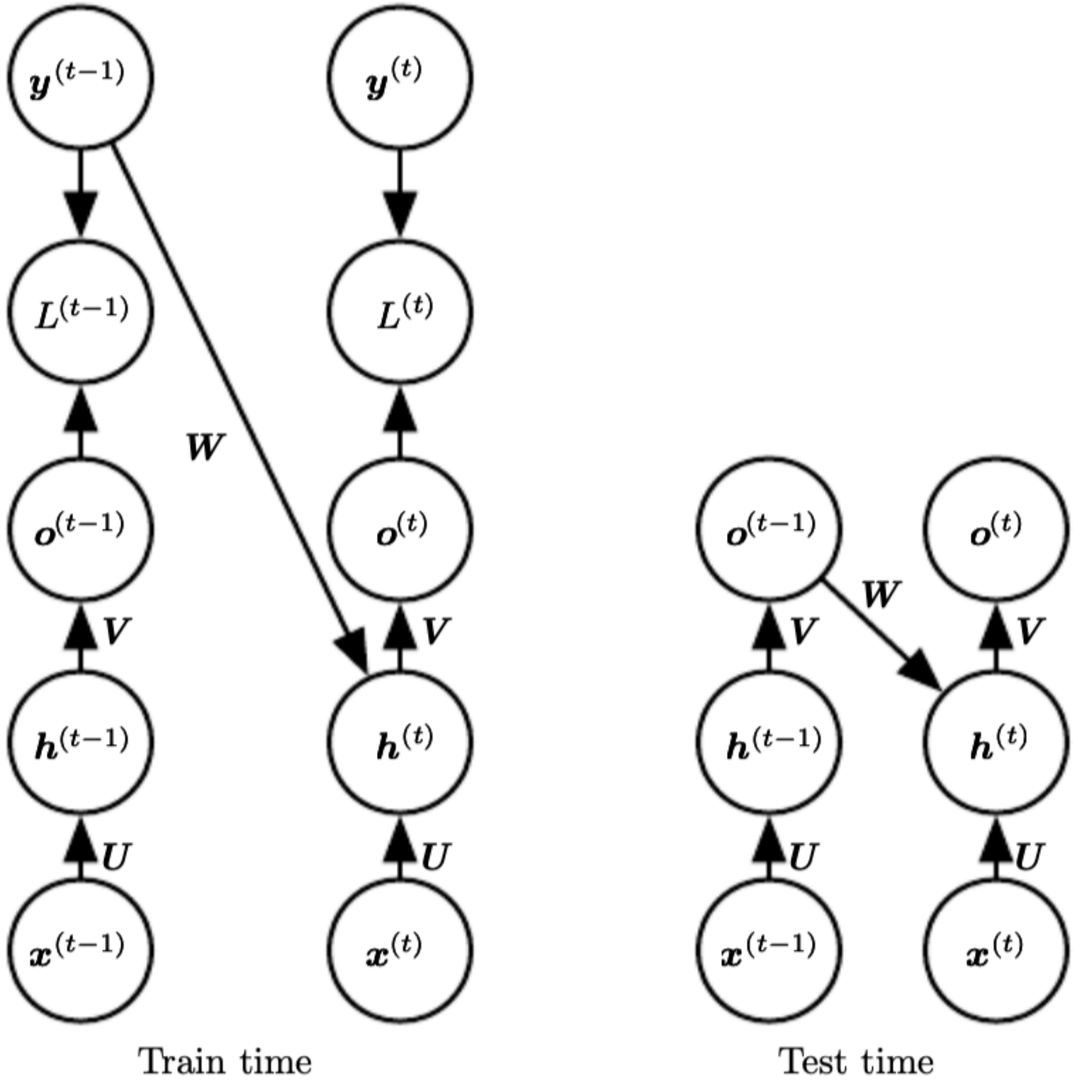
$$o^{(t)} = Vh^{(t)} + c$$

$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$



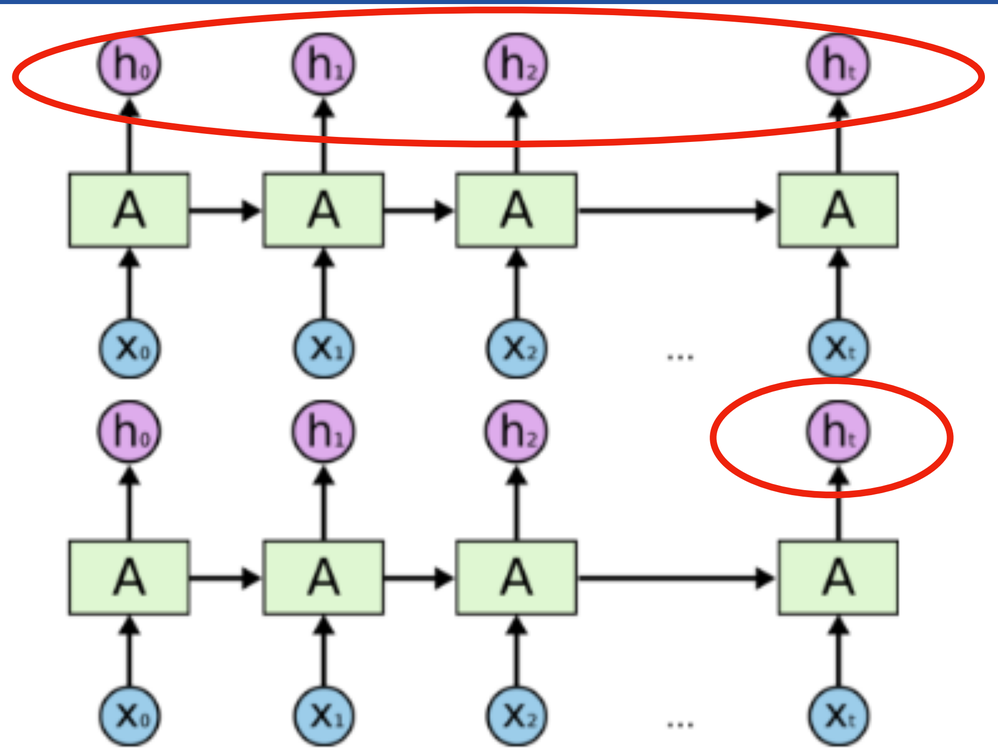
Pros and Cons of Jordan RNNs

- *Training is faster, since it can be paralleled*
- *Intrinsically less powerful*
  - *Reduced complexity due to lack of hidden-to-hidden layer*
  - *The output has to encode at once the target ground truth and the historical memory of the input (encapsulated in the hidden unit for Elamn's RNNs)*
- *Issues solved with trick called teacher forcing*
  - *Inject the label (not the output) of previous t as input at training time*



RNN in practice

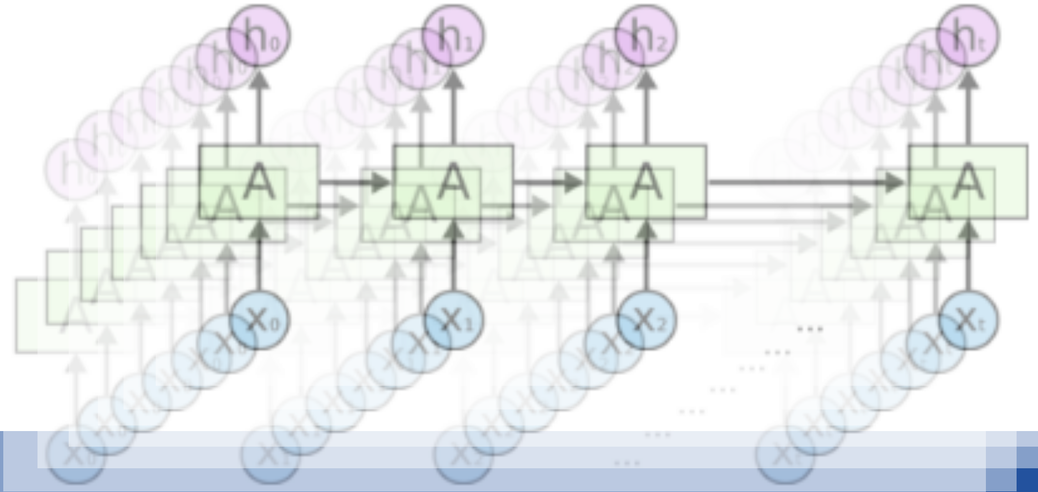
- *Not always one is interested to have a sequence as output*
  - *One could just focus on the result of the last iteration, translating a sequence in a quantity. The network is a lossy summary of the sequence (many values compressed to one output)*



- **return_sequences**: Boolean. Whether to return the last output in the output sequence, or the full sequence. Default: `False`.

From Keras

- *One typically operates many recurrent units at once in a recurrent layer (like nodes in a dense layer, or kernels for CNN)*

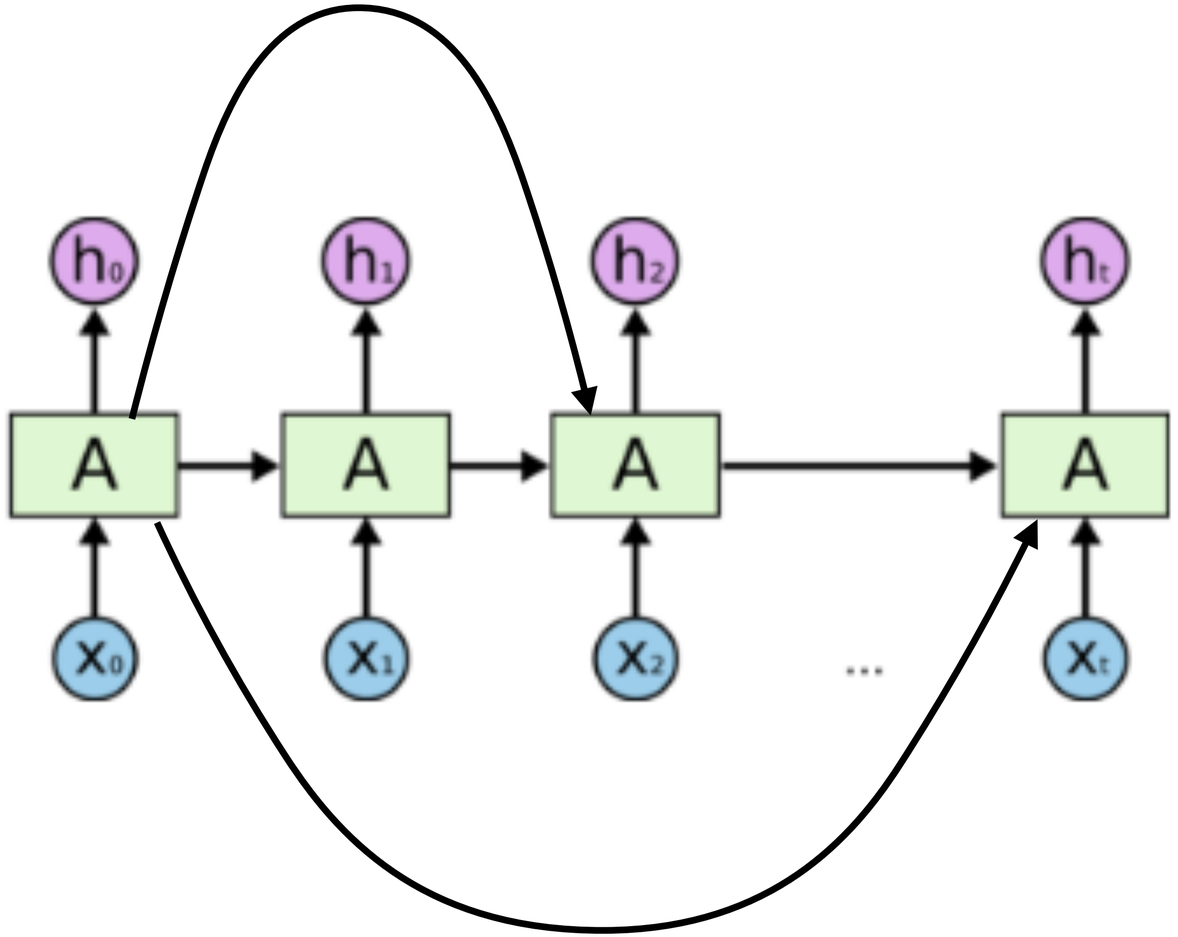


Challenges with RNNs

- *Particularly for long chains of data, the greatest challenge is to remember information across a large number of steps*
- *There is fundamentally a gradient stability issue:*
  - *In applying the same set of operations multiple times, same gradient appears multiplied many times when computing the total gradient*
  - *If gradient > 1 (<1), multiple powers would give exploding (vanishing) gradient, making the learning progress unstable*
- *There are three ways to keep long term memory “by hand” w/o gradient instabilities*
  - *Skip connections*
  - *Leaky connection*
  - *Removing connections*

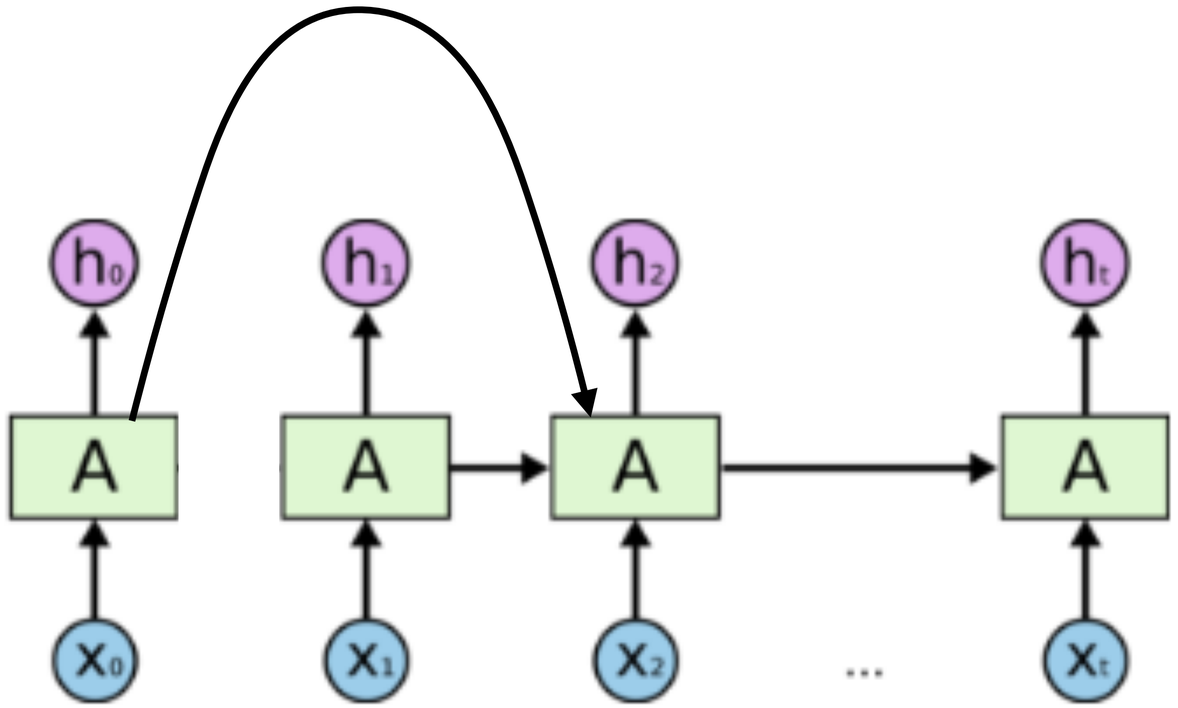
Skip Connections

- *Use direct links from one part to another to let information flow (also) skipping intermediate steps*
  - *Skip connection from t to t+d (d = delay) slows gradient vanishing/ explosion by factor 1/d.*
  - *Skipping multiple steps help with gradient convergence: it guarantees that a given step can influence the learning d steps down the chain (by-hand longer memory)*



Removing Connections

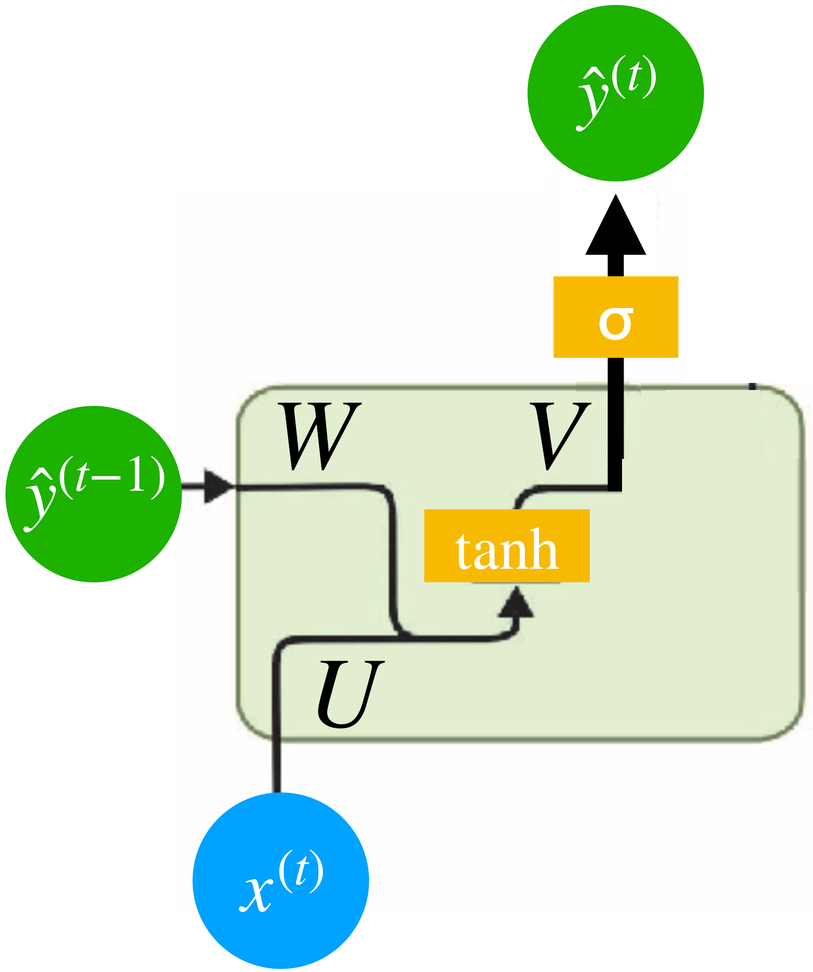
- *Similar to skip connection*
  - *In this case, though, the short-term connection is removed when the long-term connection is added*
  - *This forces the propagation of the information on many steps, since the step-by-step put is cut*



Leaky Connection

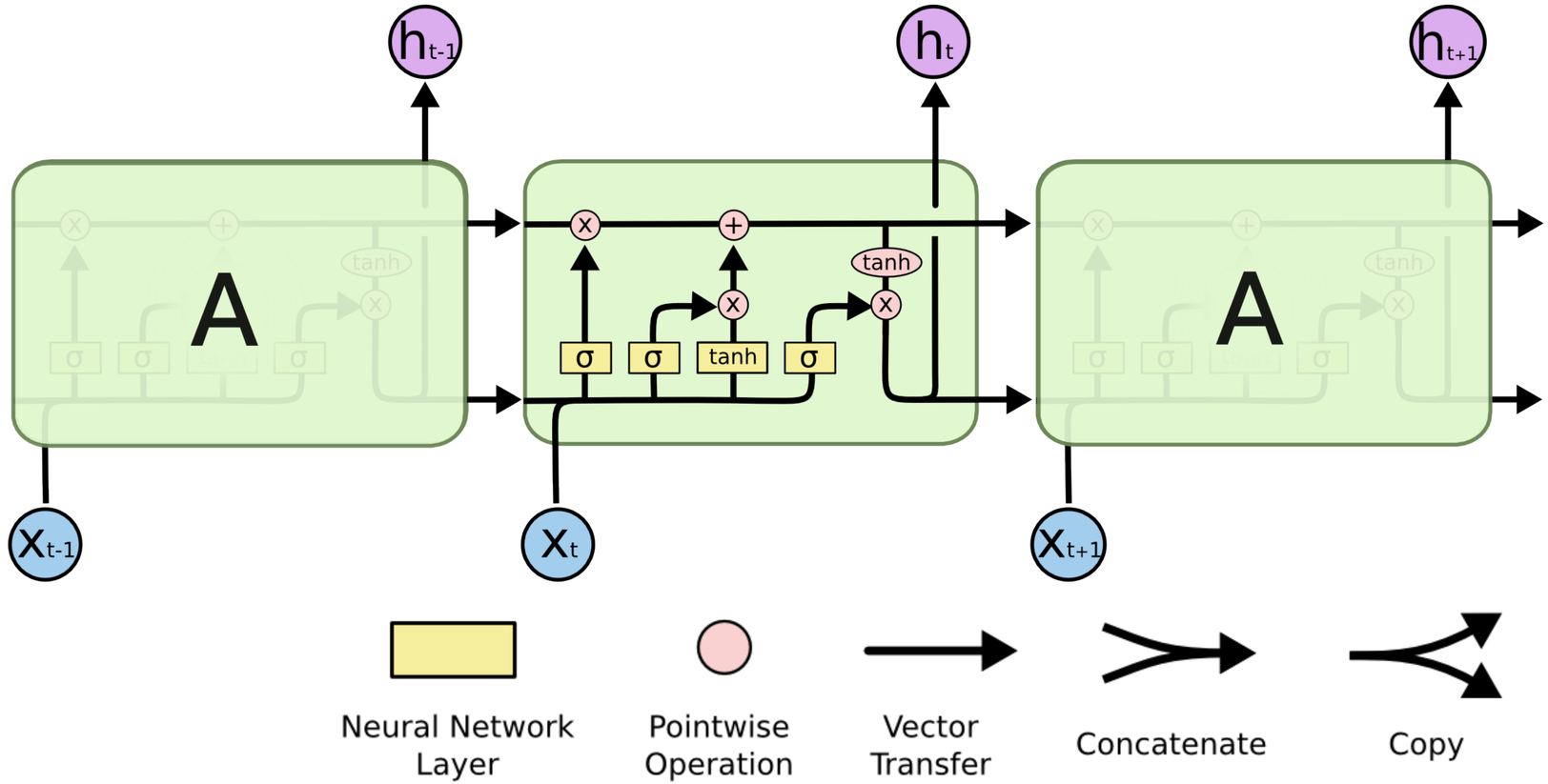
one can use a tenable parameter to decide the relative importance of the past info and the new one

- *if a=1, $y^{(t-1)}$ flows directly to $y^{(t)}$ (skip connection)*
- *if a=0, $y^{(t)}$ comes from the processing of $x^{(t)}$ (no connection)*
- *Intermediate values gives intermediate situations (and intermediate scales between short and long term)*



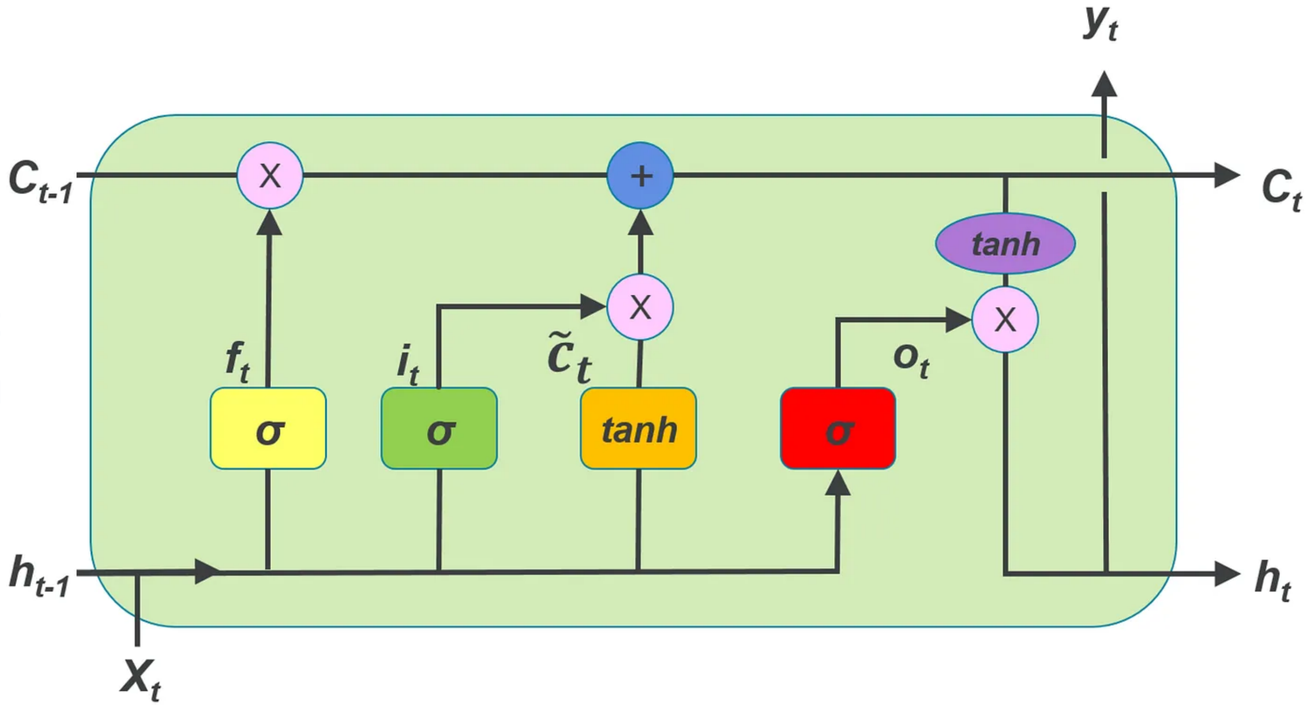
Long Short Memory Networks

- *Long-Short Memory Networks (LSTMz) are a more complicated version of RNNs*
  - *Gate mechanism: additional hidden-to-hidden connection playing the role of long-term memory, to be kept or released depending on threshold*



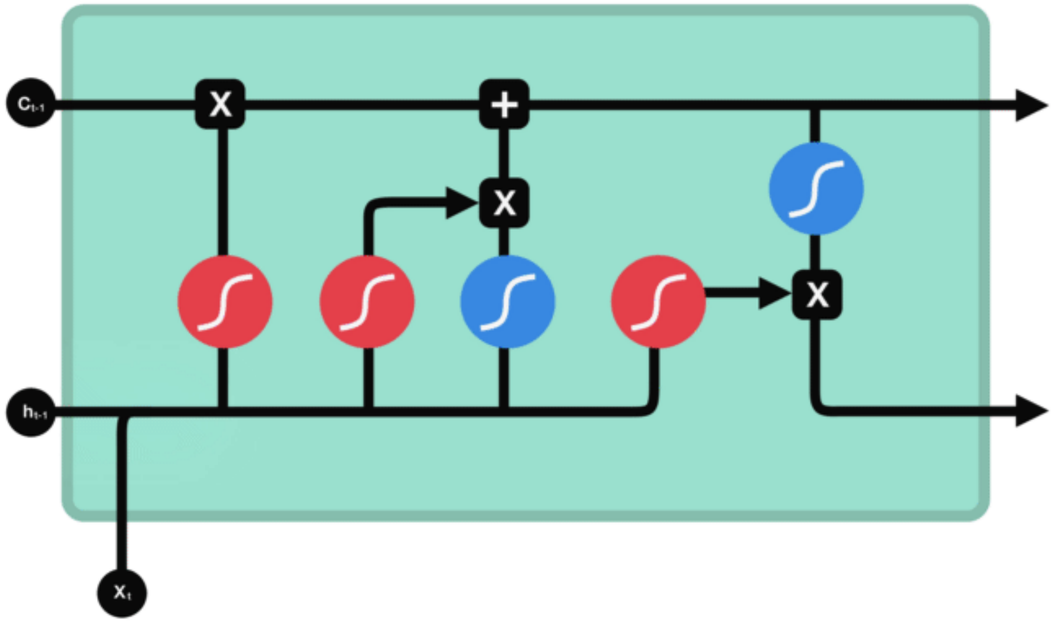
Long Short Memory Networks

- *Long-Short Memory Networks (LSTMs) are a special kind of RNN in which the information flows in two ways*
  - *through a hidden state (like RNN)*
  - *through a cell state, which acts as a memory and decides how much of past inference is carried over*
- *The memory pass/release happens through gates, implemented as sigmoid functions of the input* $x^{(t)}$ *and the hidden state* $h^{(t-1)}$



LSTM processing

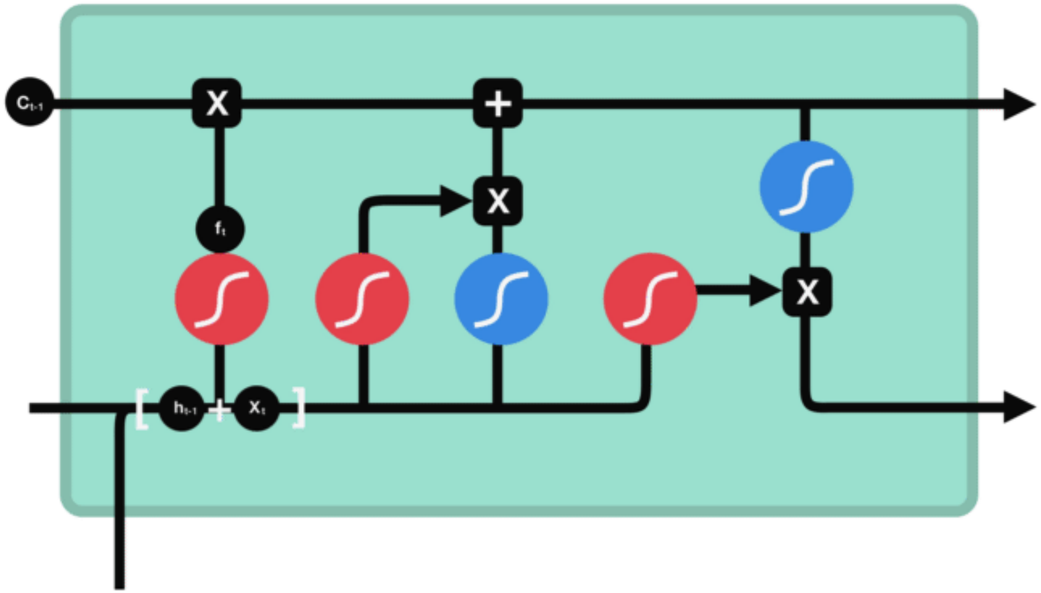
- ***Forget gate:*** *decide which amount of information should be kept*
  - *yes/no question -> sigmoid function*
  - *decision taken based on context and input*



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

LSTM processing

- ***Input gate:***
- *As before, logit given by context and input*
  - *Normalized in (-1, 1) by a tanh*
  - *Gated by a sigmoid*
  - *The sigmoid decides which fraction of the manipulated input is passed to the cell*

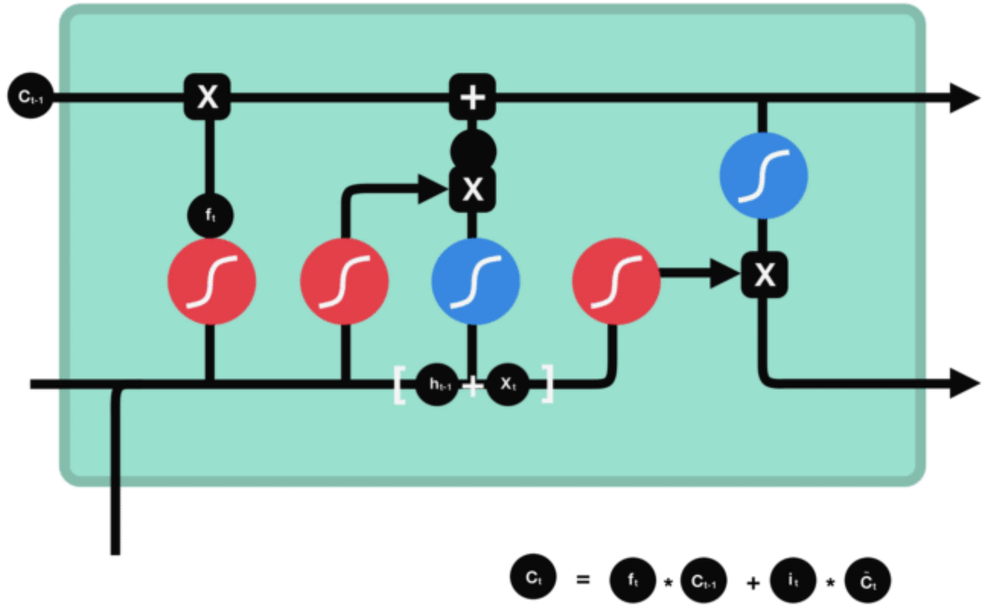


$$i^{(t)} = \sigma(U^i x^{(t)} + W^i h^{(t-1)} + b^i)$$

$$\tilde{c}^{(t)} = \tanh(U^c x^{(t)} + W^c h^{(t-1)} + b^c)$$

LSTM processing

- ***Cell State:***
- *The previous cell is multiplied by the cell gate to see which fraction survives*
- *Then we (point wise) add it to what comes out of the input gate*
  - *This is the new cell state*
    - *Passed to next time step*
    - *Used to updated the hidden state*



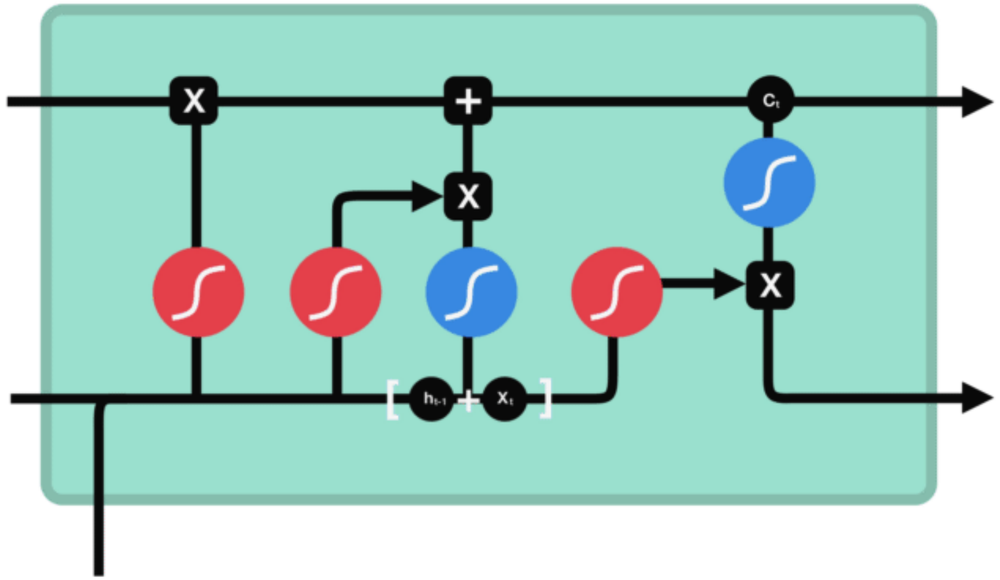
C_{t-1} previous cell state
f_t forget gate output
i_t input gate output
$\tilde{C}_t$ candidate
C_t new cell state

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

$$c^{(t)} = c^{(t-1)} f^{(t)} + \tilde{c}^{(t)} \cdot i^{(t)}$$

LSTM processing

- ***Output gate:***
- *As before, logit given by context and input*
- *Gated by a sigmoid (as for cell and forget gates)*
- *The updated hidden state is the product of the output gate and the tanh of the cell state*



C_{t-1} previous cell state
f_t forget gate output
i_t input gate output
$\tilde{C}_t$ candidate
C_t new cell state
o_t output gate output
h_t hidden state

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

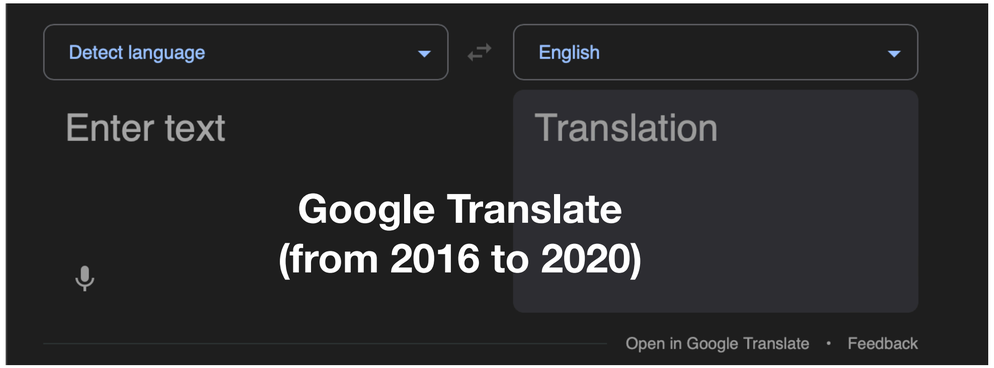
$$h^{(t)} = o^{(t)} \cdot \tanh c^{(t)}$$

What is going on

- *Unlike RNNs, LSTM controls the flow of information from one step to the other, using the gates*
  - *By erasing part of the cell memory, it limits the concatenation of large or small numbers when computing gradients, hence the vanishing/exploding gradient problem*
  - *In this way, this network adds long-term memory capability to the short-term memory of a traditional RNN*

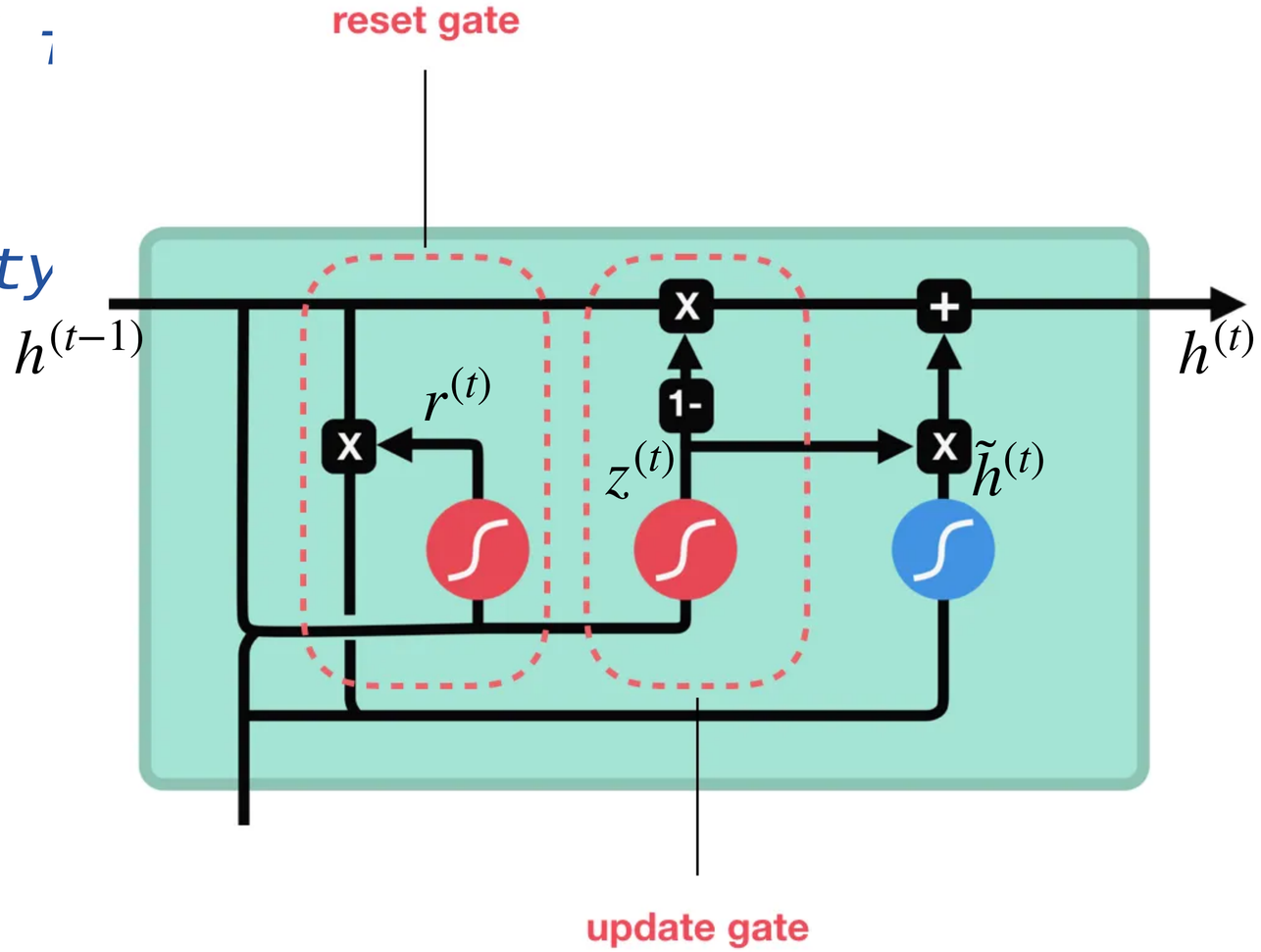
History of LSTMs

- *Introduced in 1995 by Hochreiter and Schmidhuber*
  - *Outperformed other options and became the default RNN*
  - *Improved with time adding variants with a few changes*
    - *Notably, the concept of bidirectional LSTM, processing the sequence in direct and inverted order*
  - *Around 2006, BiDirectional LSTM revolutionise speech recognition and text-to-speech (virtual assistants, e.g. Google and Amazon)*
    - *Applied in many domains*
  - *Other architectures emerged (see GRU in the next slide) that matched but never really surpassed LSTM performances. After 30 years, LSTMs are still RNN state of the art*



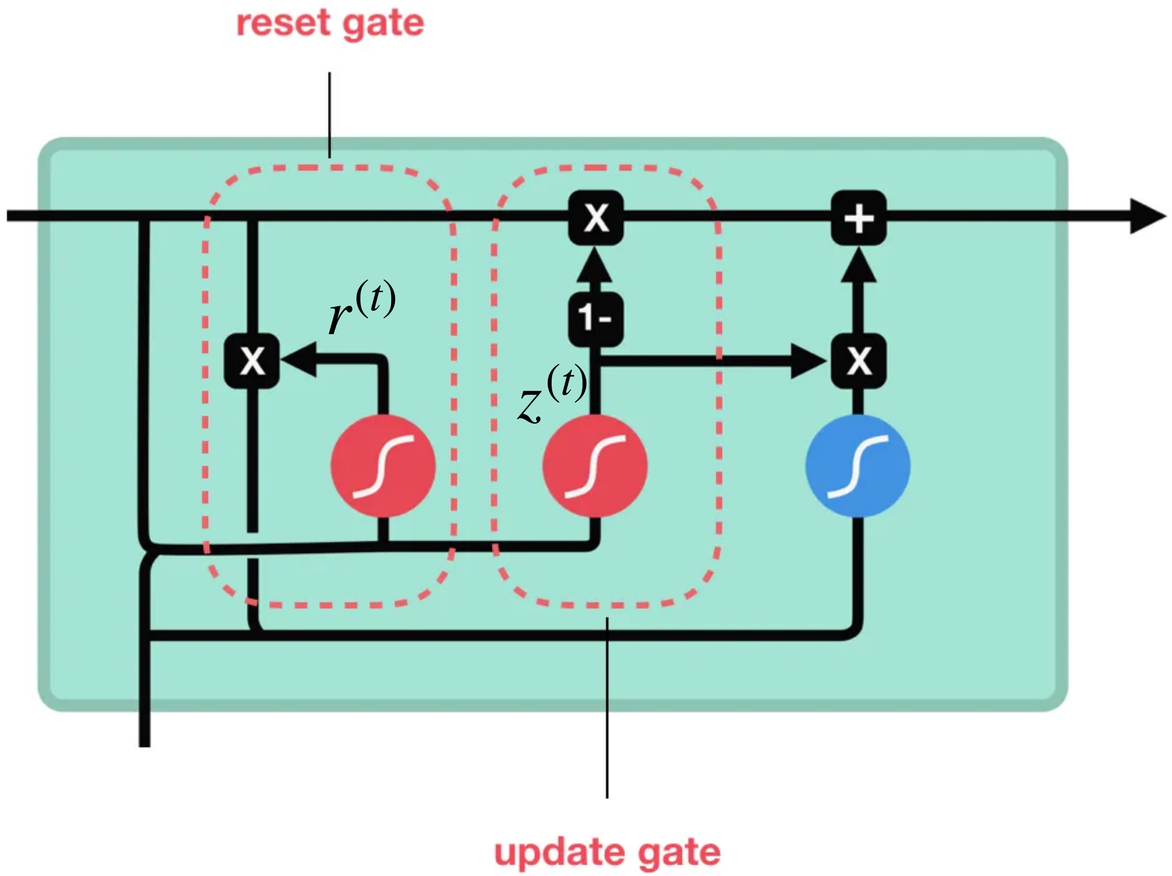
Gated Recurrent Unit

- Gated Recurrent Units (GRUs) are alternative to LSTM introduced i 2014 by Kyunghyun Cho et al.
  - Similar to LSTM for complexity but no cell state (single hidden-to-hidden connection)
  - Based on two gates
    - Reset gate: rules memory cleaning
    - Update gate: rules memory updating



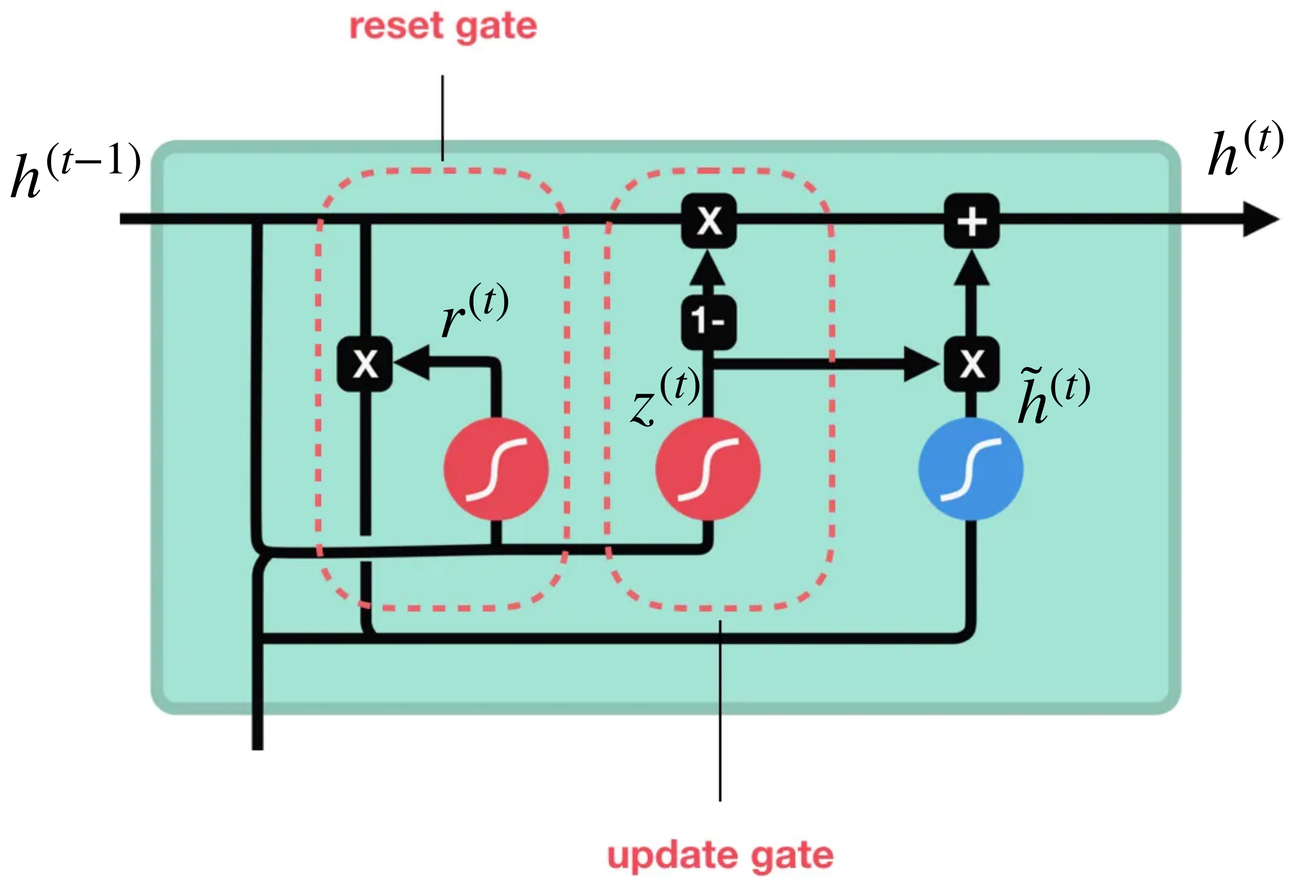
GRU processing

- ***Reset Gate:*** *update gate*
  - *combine input* $x^{(t)}$ *and context* $h^{(t-1)}$*, weighted by U and W*
  - *apply gate with sigmoid*
- ***Update Gate:*** *update gate*
  - *combine input* $x^{(t)}$ *and context* $h^{(t-1)}$*, weighted by U and W*
  - *apply gate with sigmoid*



$$r^{(t)} = \sigma(U^r x^{(t)} + W^r h^{(t-1)})$$
$$z^{(t)} = \sigma(U^z x^{(t)} + W^z h^{(t-1)})$$

GRU processing



- ***Hidden state update:***
  update gate

$$\tilde{h}^{(t)} = \tanh(Ux^{(t)} + W(r^{(t)} \cdot h^{(t-1)}))$$

$$h^{(t)} = (1 - z^{(t)})h^{(t-1)} + z^{(t)}\tilde{h}^{(t)}$$

RNNs in Transformer Era

- *Until 2020, advanced RNNs (bidirectional LSTMs and GRUs) were the default option for serial data*
- *In 2020 transformers changed this picture*
  - *No sequential processing, faster training*
- *Last year, revival of RNN with the Mamba model, a so-called State Space Model (SSM)*
  - *Similar to GRU, but structured to that $h^{(t)}$ is a linear function of $h^{(t-1)}$. This makes the training faster*
  - *Mamba outperformed transformers in handling long sequences (up to 1M steps) with faster inference, scaling linearly with sequence length (transformer is quadratic)*
  - *We are not covering SSMs, but keep in mind that RNN are not dead*