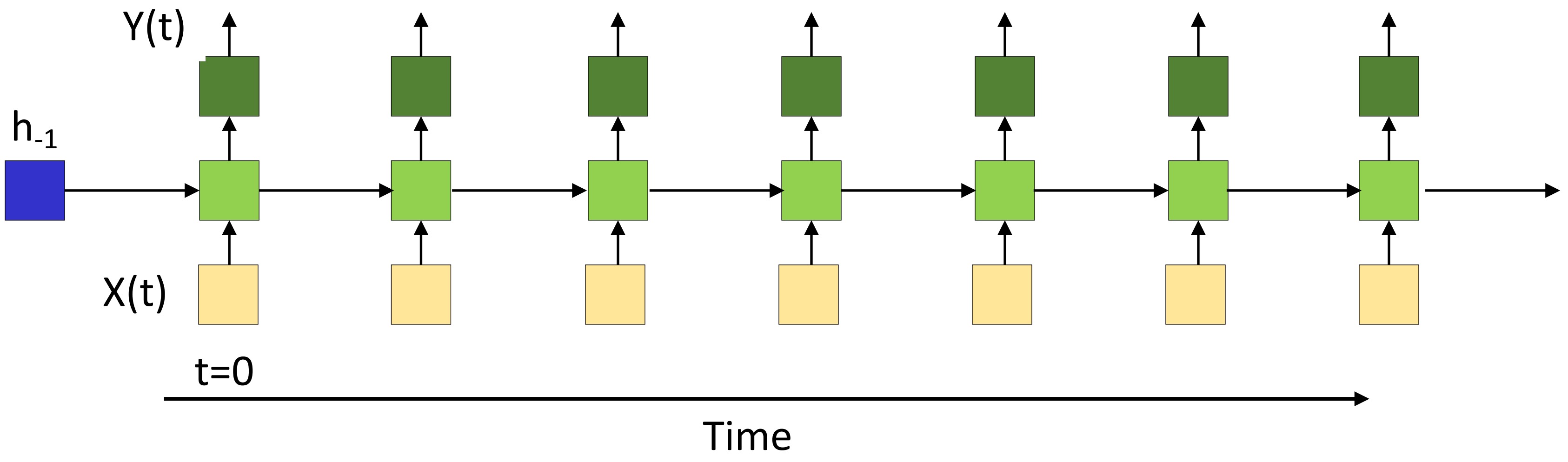
Recurrent Neural Networks: Stability analysis and LSTMs

M. Soleymani
Sharif University of Technology
Spring 2025

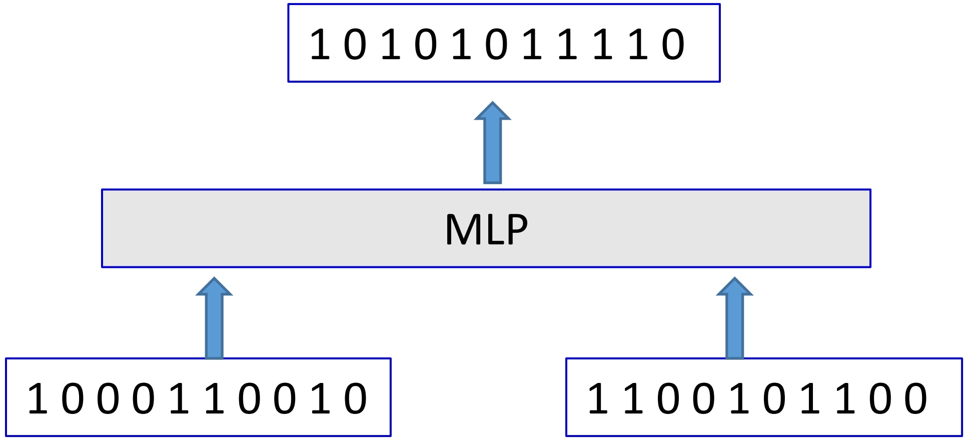
Most slides have been adopted from Bhiksha Raj, 11-785, CMU 2019
and some from Fei Fei Li and colleagues lectures, cs231n, Stanford 2022

Story so far



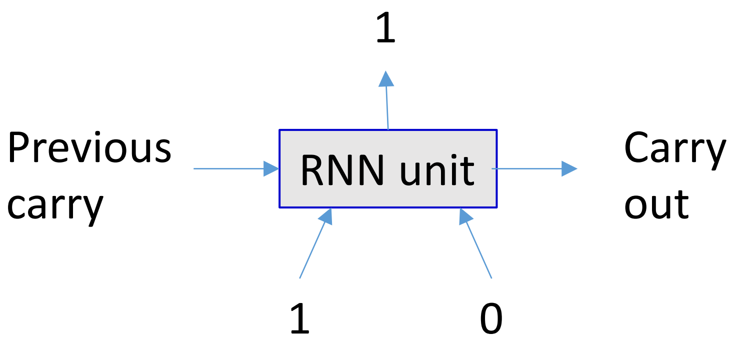
- ***Recurrent structures*** are good for analyzing time series data with ***long-term*** dependence on the past
  - These are ***recurrent*** neural networks

Recurrent structures can do what static structures cannot



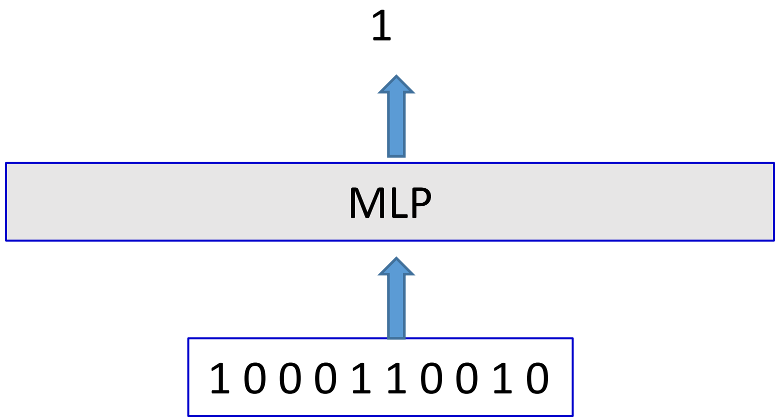
- The addition problem: Add two N-bit numbers to produce a N+1-bit number
  - Input is binary
  - Will require large number of training instances
    - Output must be specified for every pair of inputs
    - Weights that generalize will make errors
  - Network trained for N-bit numbers will not work for N+1 bit numbers

MLPs vs RNNs



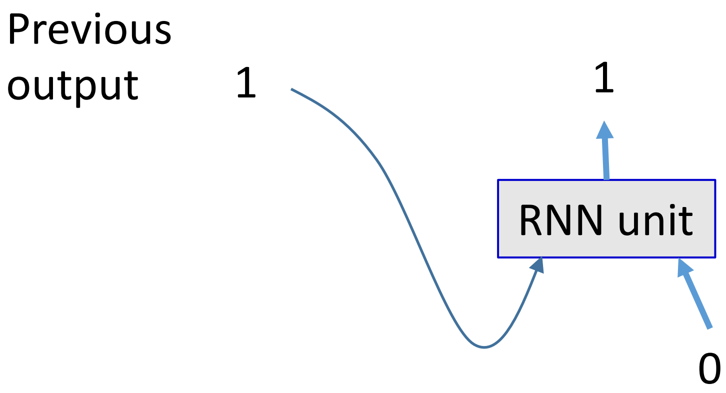
- The addition problem: Add two N-bit numbers to produce a N+1-bit number
- **RNN solution:** Very simple, can add two numbers of any size

MLP: The parity problem



- Is the number of “ones” even or odd
- Network must be complex to capture all patterns
  - XOR network, quite complex
  - Fixed input size

RNN: The parity problem



- Trivial solution
- Generalizes to input of any size

RNNs..

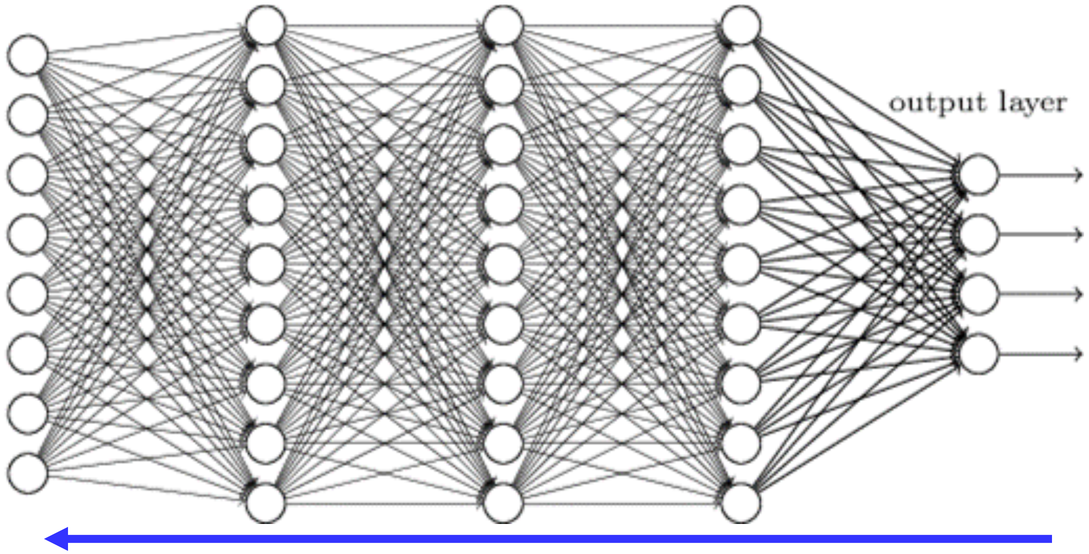
- Excellent models for time-series analysis tasks
  - Time-series prediction
  - Time-series classification
  - Sequence prediction..
  - They can even simplify problems that are difficult for MLPs

- But the memory isn’t all that great..
  - Also..

The vanishing gradient problem

- A particular problem with training deep networks..
  - Any deep network, not just recurrent nets
  - The gradient of the error with respect to weights is unstable..

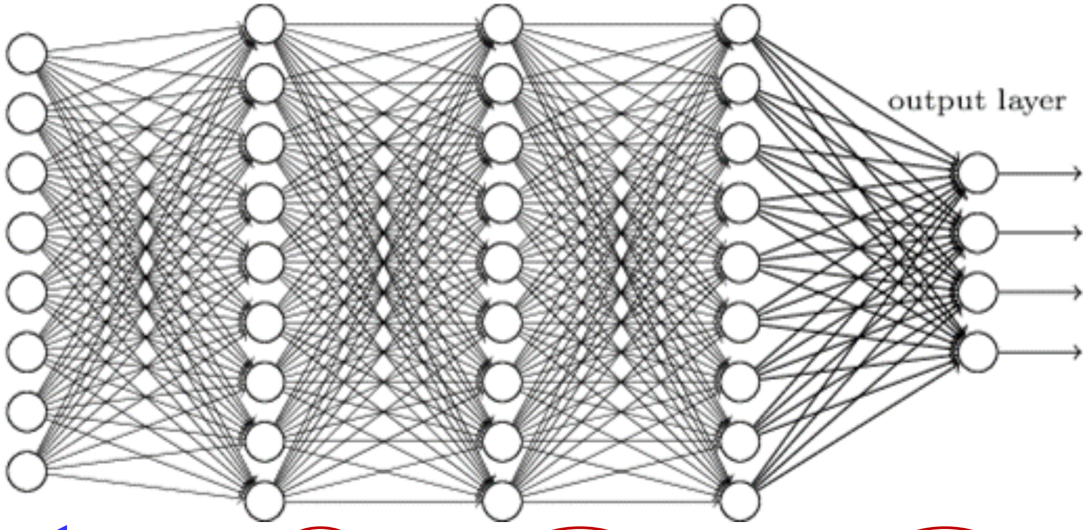
Reminder: Gradient problems in deep networks



$$\nabla_{f^{[l]}} Loss = \nabla_{f^{[L]}} Loss . \nabla f^{[L]} . W^{[L]} . \nabla f^{[L-1]} . W^{[L-1]} \ldots \nabla f^{[l+1]} . W^{[l+1]}$$

- The gradients in the lower/earlier layers can *explode* or *vanish*
  - Resulting in insignificant or unstable gradient descent updates
  - Problem gets worse as network depth increases

Reminder: Training deep networks



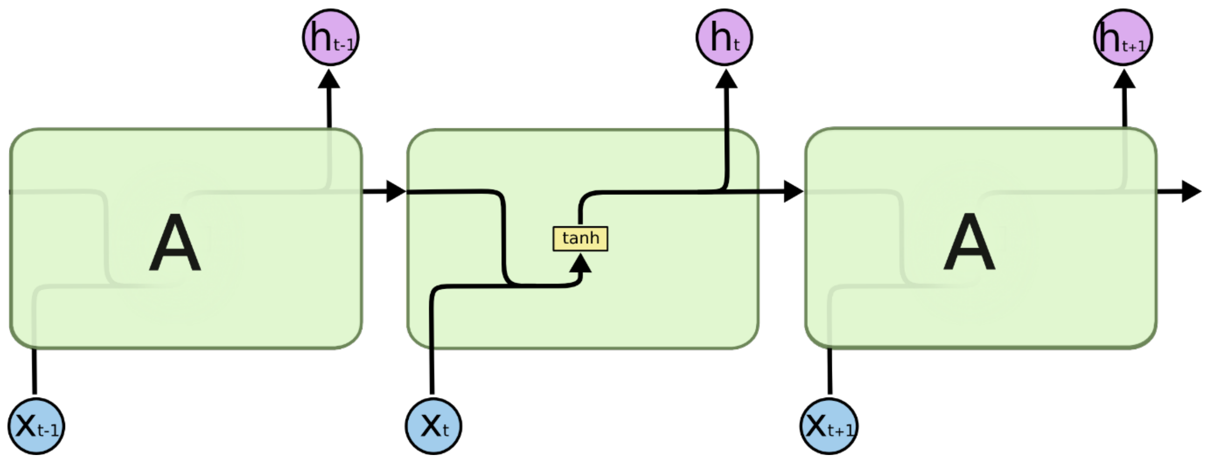
$$\nabla_{f^{[l]}} Loss = \nabla_{f^{[L]}} Loss . \nabla f^{[L]} . W^{[L]} . \nabla f^{[L-1]} . W^{[L-1]} \ldots \nabla f^{[l+1]} . W^{[l+1]}$$

- As we go back in layers, the Jacobians of the activations constantly *shrink* the derivative
  - After a few layers the derivative of the loss at any time is totally “forgotten”

The long-term dependency problem

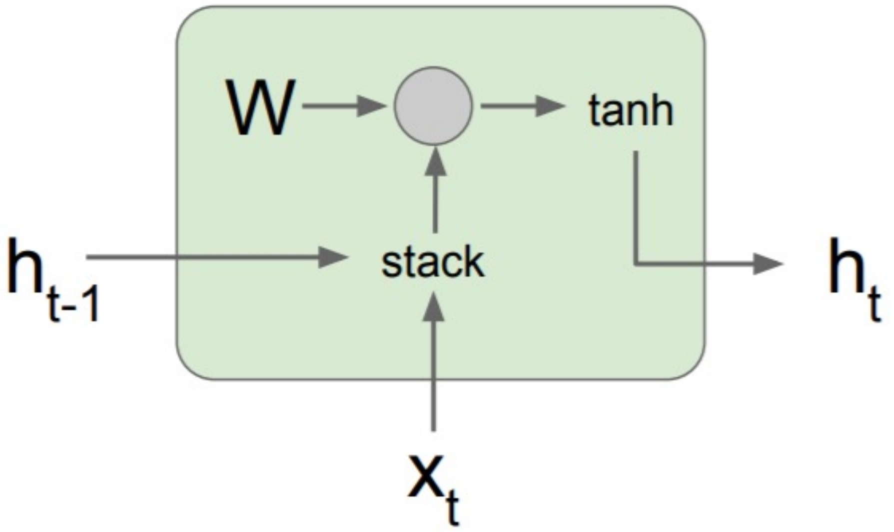
- Must know to “remember” for extended periods of time and “recall” when necessary
  - Can be performed with a multi-tap recursion, but how many taps?
  - Need an alternate way to “remember” stuff

Standard RNN



- Recurrent neurons receive past recurrent outputs and current input as inputs
- Processed through a tanh() activation function
  - As mentioned earlier, tanh() is the generally used activation for the hidden layer
- Current recurrent output passed to next higher layer and next time instant

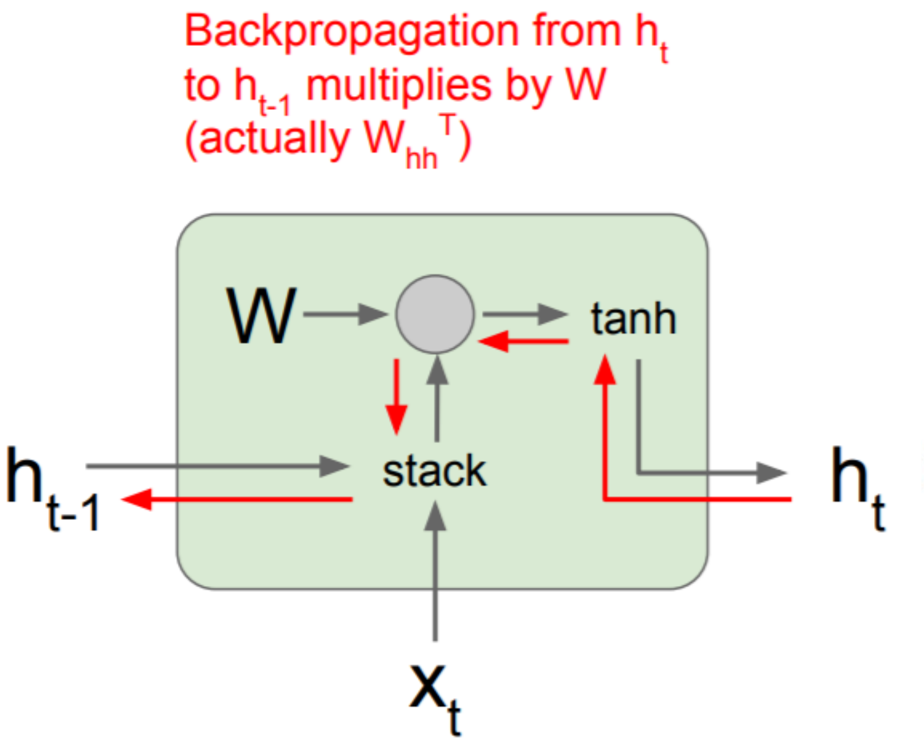
Vanilla RNN Gradient Flow



$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$
$$= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$
$$= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013

Vanilla RNN Gradient Flow



$$
\begin{aligned}
h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\
&= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right) \\
&= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)
\end{aligned}
$$

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013

Enter the *LSTM*

- *Long Short-Term Memory*
- Explicitly latch information to prevent decay / blowup

- Following notes borrow liberally from
- http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Better units for recurrent models

- More complex hidden unit computation in recurrence!
  - $h_t = LSTM(x_t, h_{t-1})$
  - $h_t = GRU(x_t, h_{t-1})$
- Main ideas:
  - keep around memories to capture long distance dependencies
  - allow error messages to flow at different strengths depending on the inputs

Long Short Term Memory (LSTM)

Vanilla RNN

$$h_t = \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

LSTM

Four gates

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

Cell state

$$c_t = f \odot c_{t-1} + i \odot g$$

Hidden state

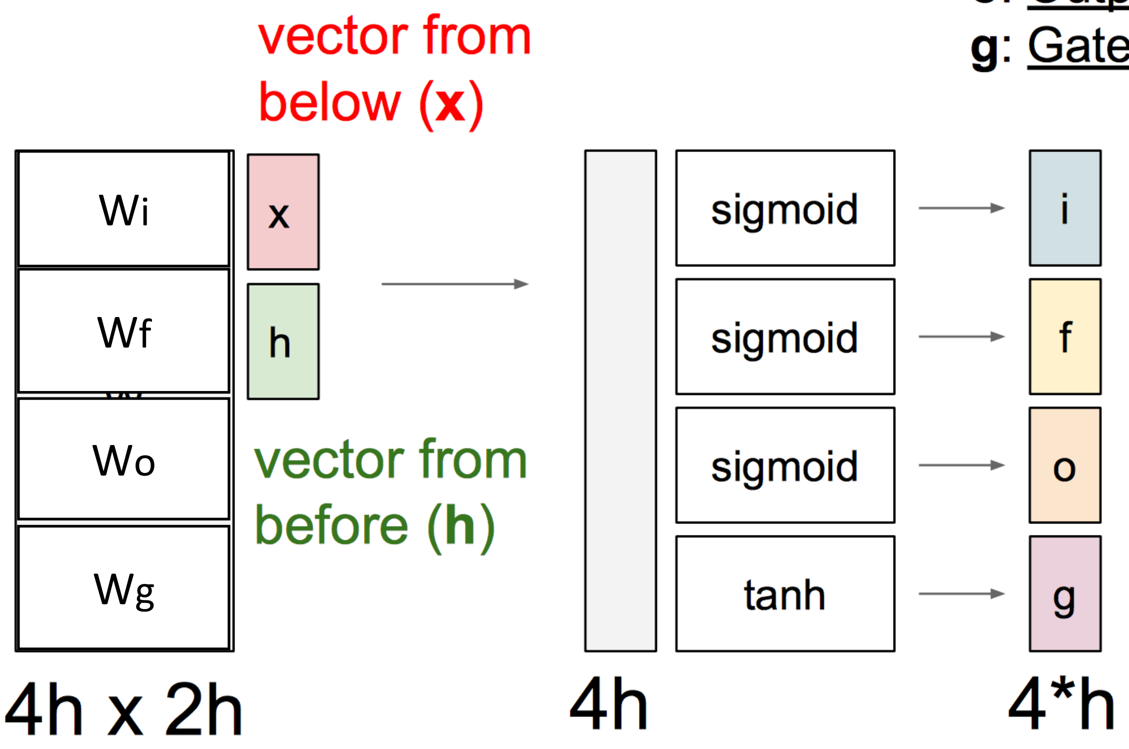
$$h_t = o \odot \tanh(c_t)$$

Hochreiter and Schmidhuber, “Long Short Term Memory”, Neural Computation 1997

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

i: Input gate, whether to write to cell
f: Forget gate, Whether to erase cell
o: Output gate, How much to reveal cell
g: Gate gate (?), How much to write to cell



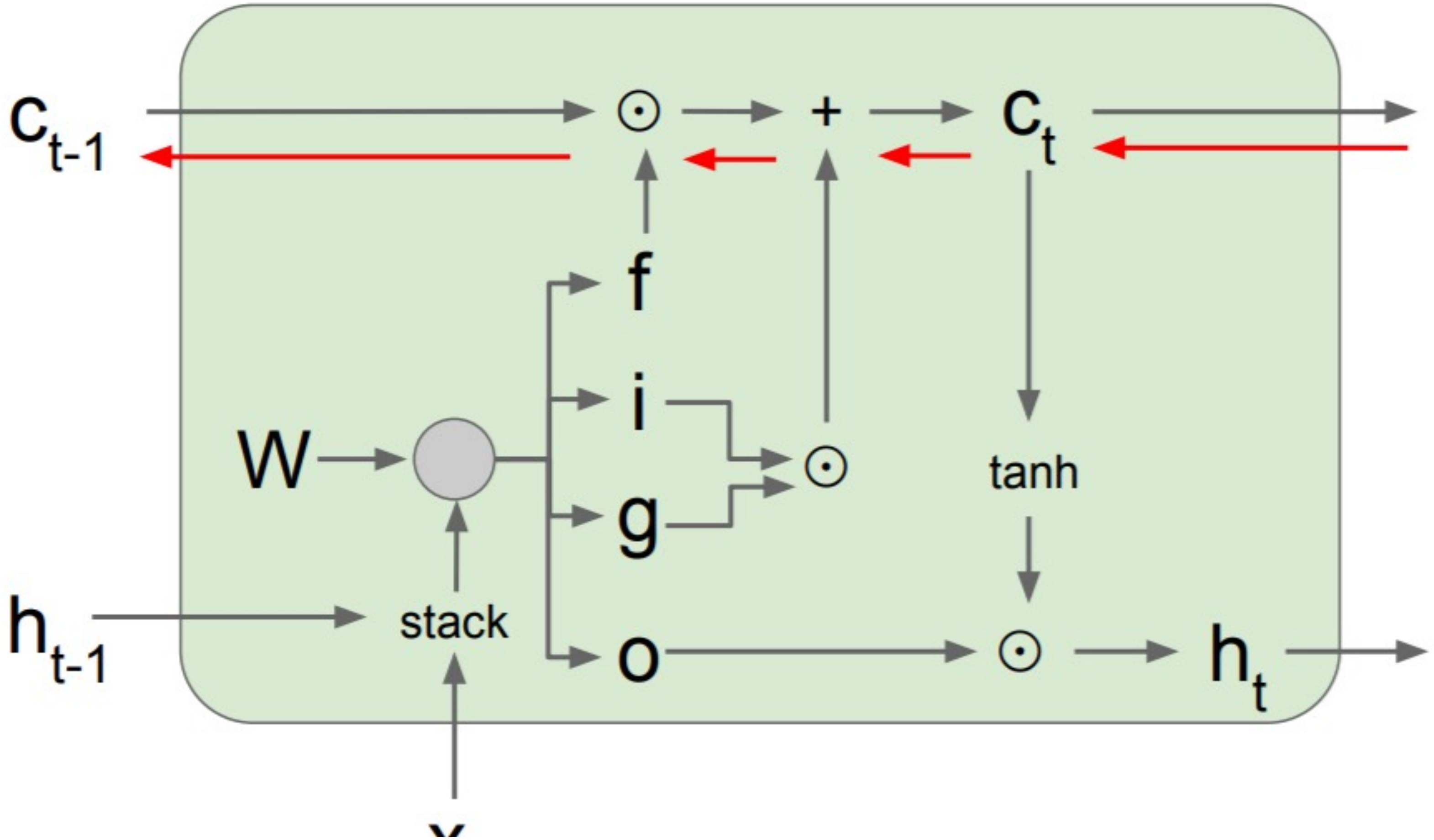
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]



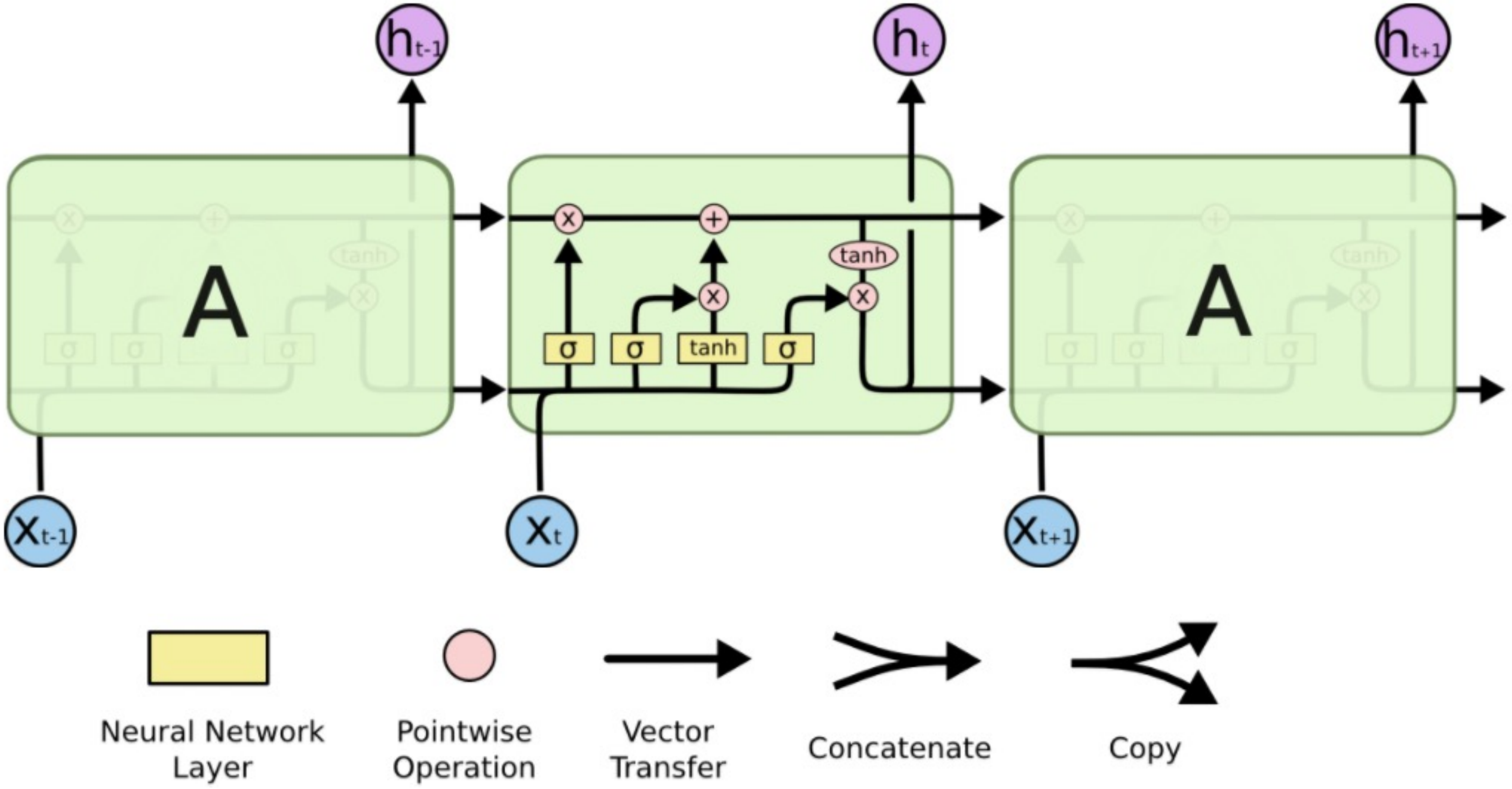
Backpropagation from c_t to c_{t-1} only elementwise multiplication by f, no matrix multiply by W

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

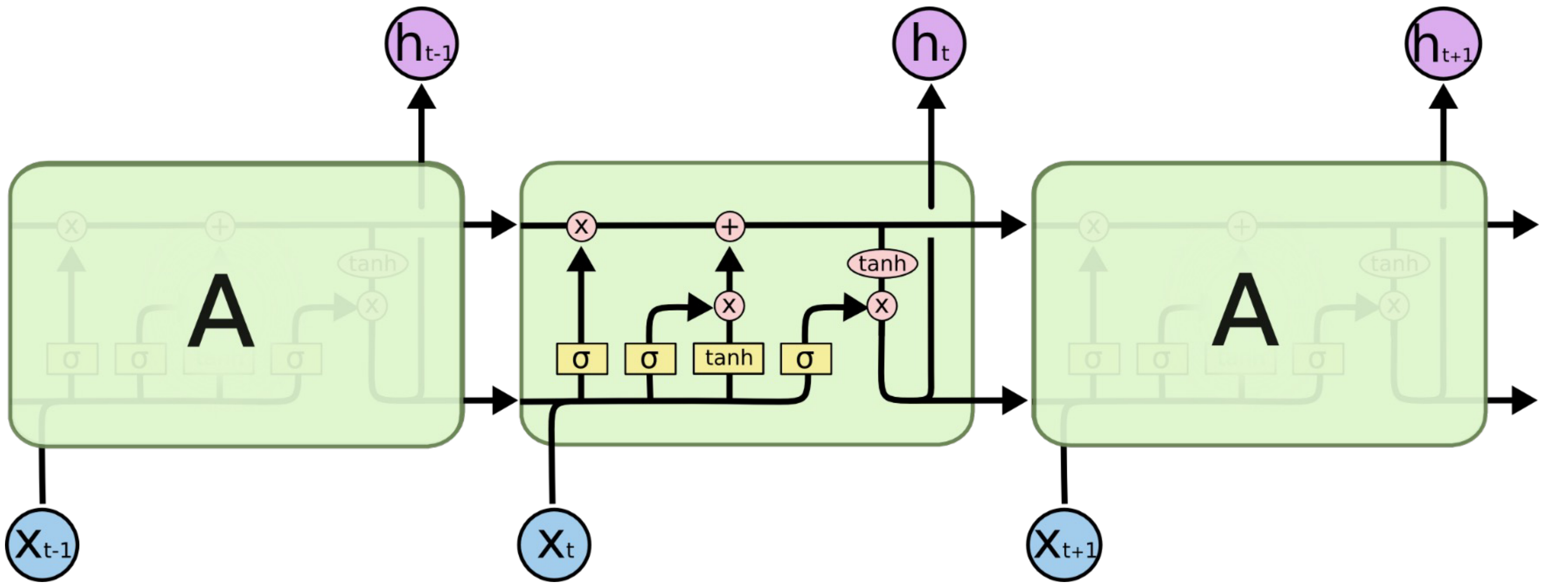
$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Some visualization

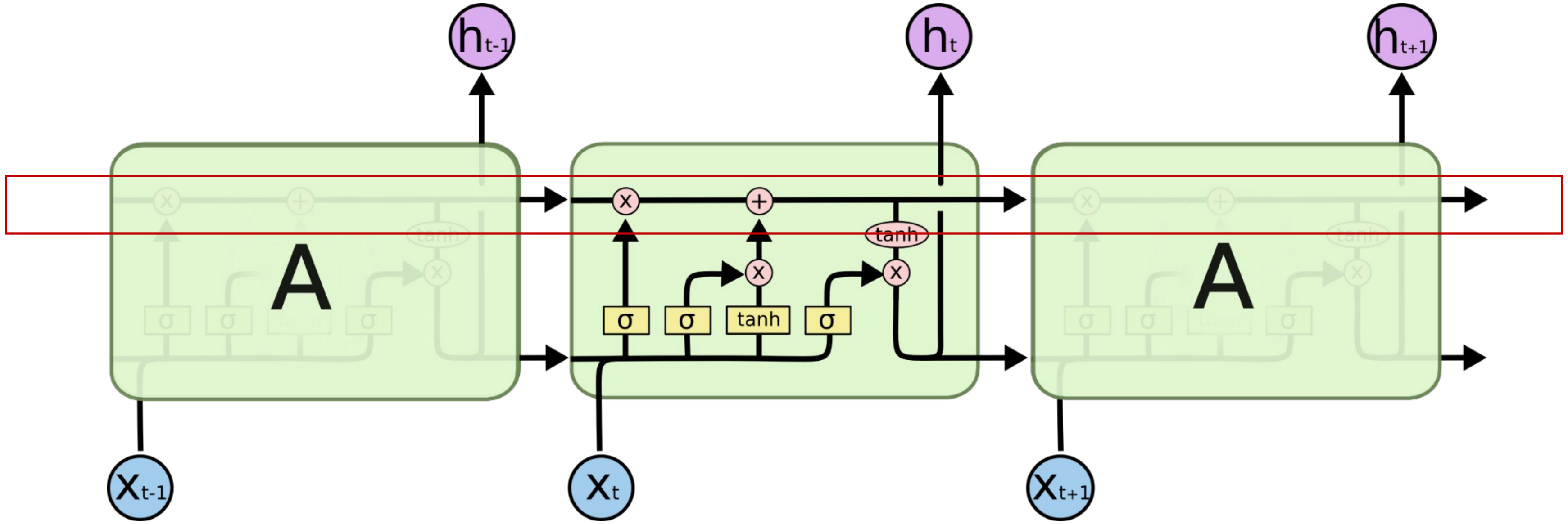


Long Short-Term Memory



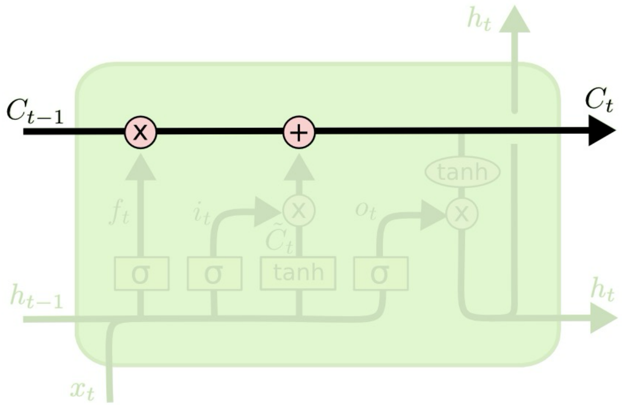
- The $\sigma()$ are *multiplicative gates* that decide if something is important or not
- Remember, every line actually represents a *vector*

LSTM: Constant Error Carousel



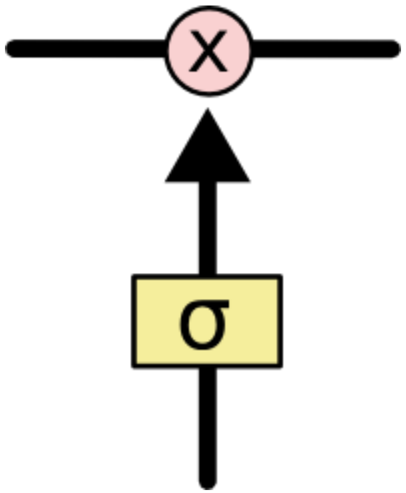
- Key component: a *remembered cell state*

LSTM: CEC



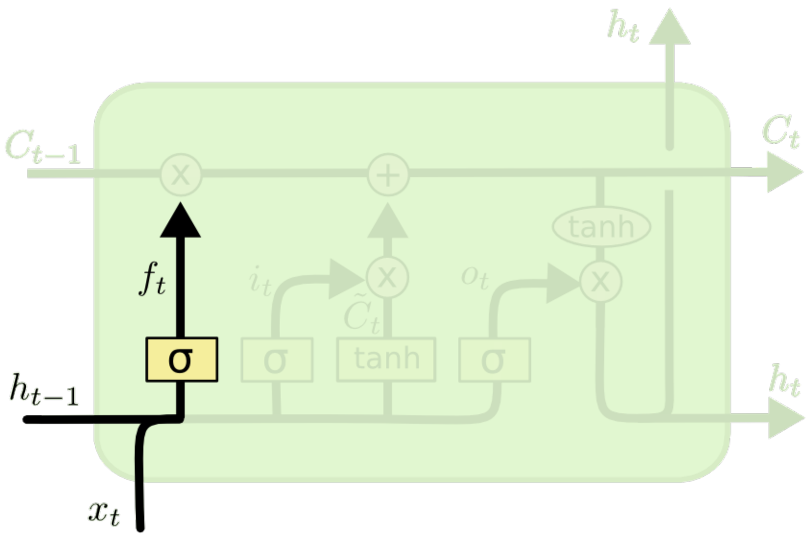
- C_t is the linear history
- Carries information through, only affected by a gate
  - And *addition of history,* which too is gated...

LSTM: Gates



- Gates are simple sigmoidal units with outputs in the range (0,1)
- Controls how much of the information is to be let through

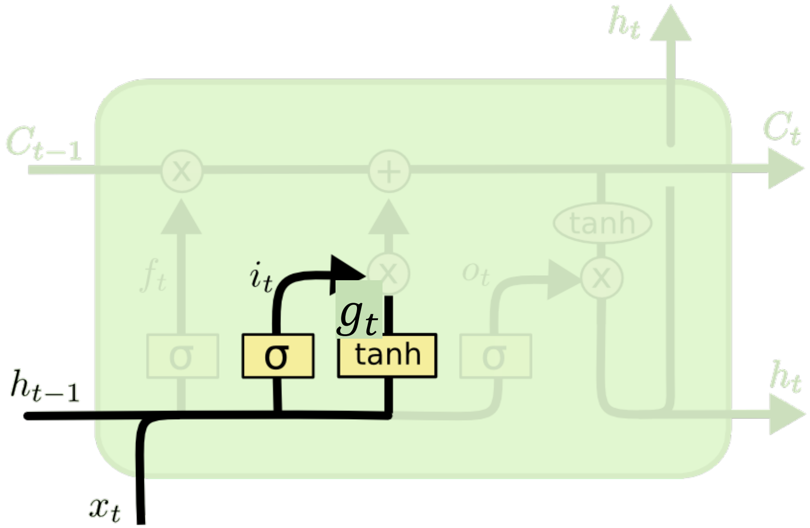
LSTM: Forget gate



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

- The first gate determines whether to carry over the history or to forget it
  - More precisely, how much of the history to carry over
  - Also called the "forget" gate
  - Note, we're actually distinguishing between the cell memory C and the state h that is coming over time! They're related though

LSTM: Input gate

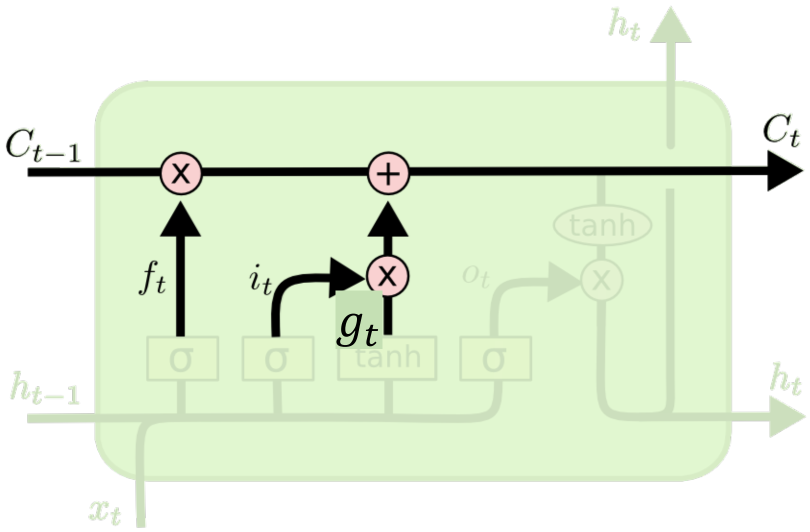


$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$

$$g_t = \tanh\left(W_g[h_{t-1}, x_t] + b_g\right)$$

- The second input has two parts
  - A perceptron layer that determines if there's something new and interesting in the input
  - A gate that decides if it's worth remembering

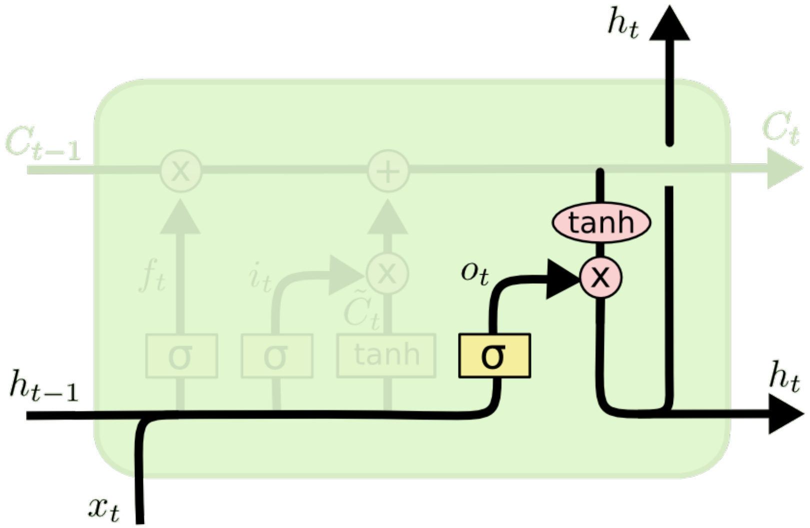
LSTM: Memory cell update



$$C_t = f_t \odot C_{t-1} + i_t \odot g_t$$

- The second input has two parts
  - A perceptron layer that determines if there's something interesting in the input
  - A gate that decides if its worth remembering
  - **If so its added to the current memory cell**

LSTM: Output and Output gate



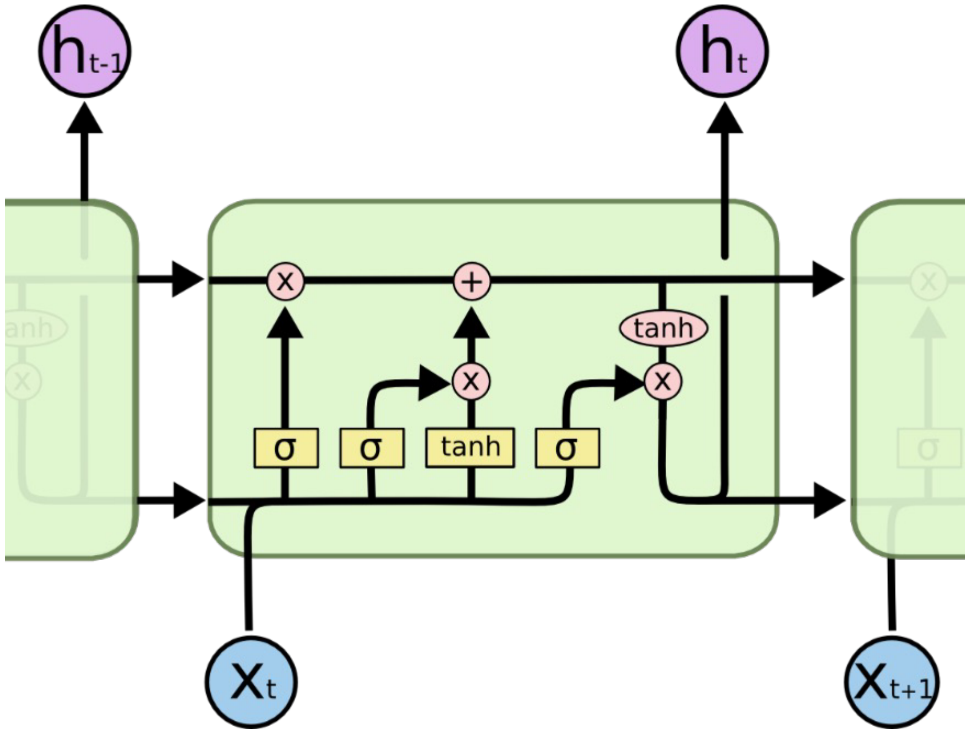
$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

- The *output* of the cell
  - Simply compress it with tanh to make it lie between 1 and -1
    - Note that this compression no longer affects our ability to *carry* memory forward
  - Controlled by an *output* gate
    - To decide if the memory contents are worth reporting at *this* time

LSTM Equations

- i_t: input gate, how much of the new information will be let through the memory cell.
- f_t: forget gate, responsible for information should be thrown away from memory cell.
- o_t: output gate, how much of the information will be passed to expose to the next time step.
- g_t: self-recurrent which is equal to standard RNN

- c_t: internal memory of the memory cell
- h_t: hidden state

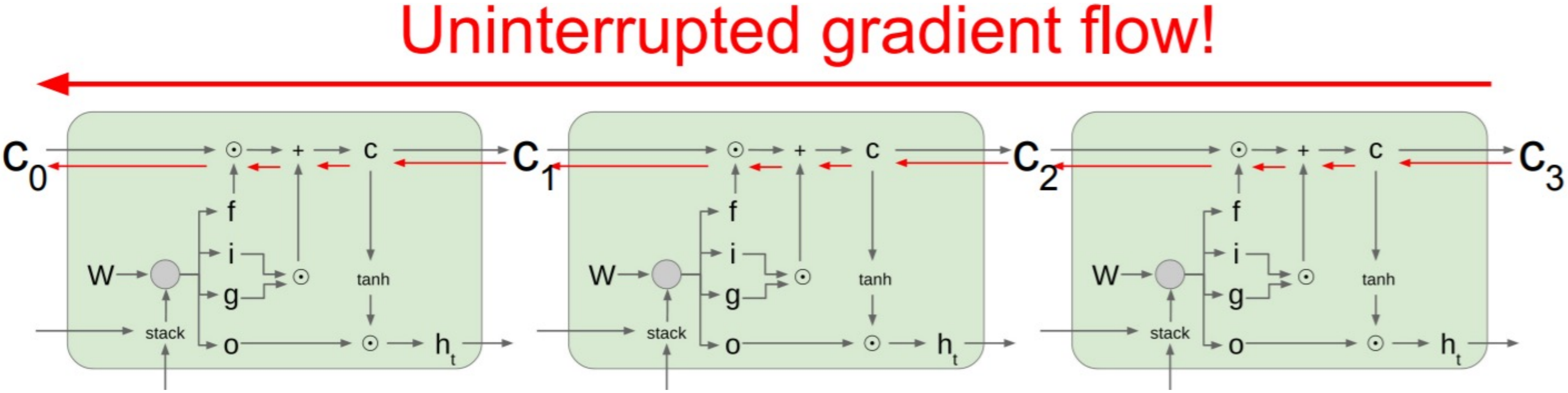


Long-short-term-memories (LSTMs)

- Gates
  - Input gate (current cell matter): $i_t = \sigma\left(W_i \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_i\right)$
  - Forget (gate 0, forget past): $f_t = \sigma\left(W_f \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_f\right)$
  - Output (how much cell is exposed): $o_t = \sigma\left(W_o \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_o\right)$
- Variables
  - New memory cell: $g_t = \tanh\left(W_c \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_c\right)$
  - Final memory cell: $C_t = i_t \circ g_t + f_t \circ C_{t-1}$
  - Final hidden state: $h_t = o_t \circ \tanh(C_t)$

Long Short Term Memory (LSTM)

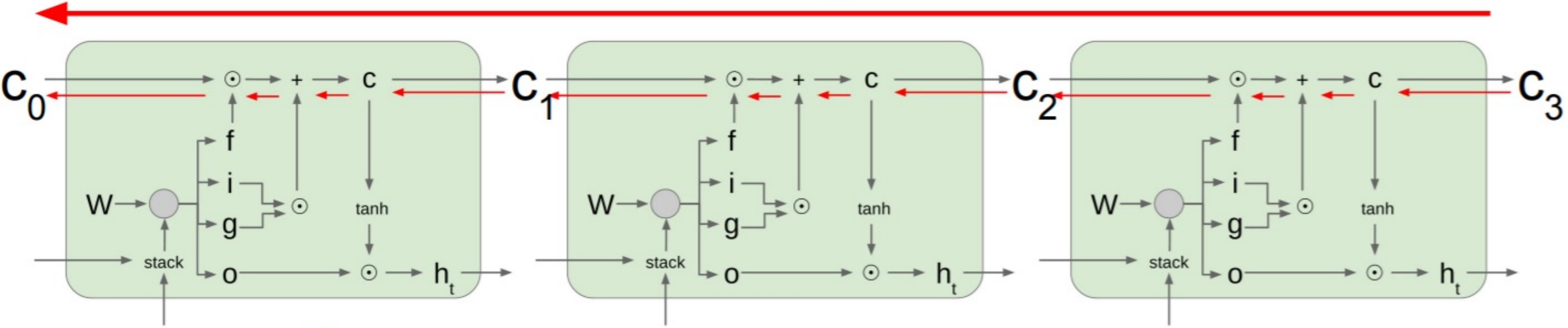
[Hochreiter et al., 1997]



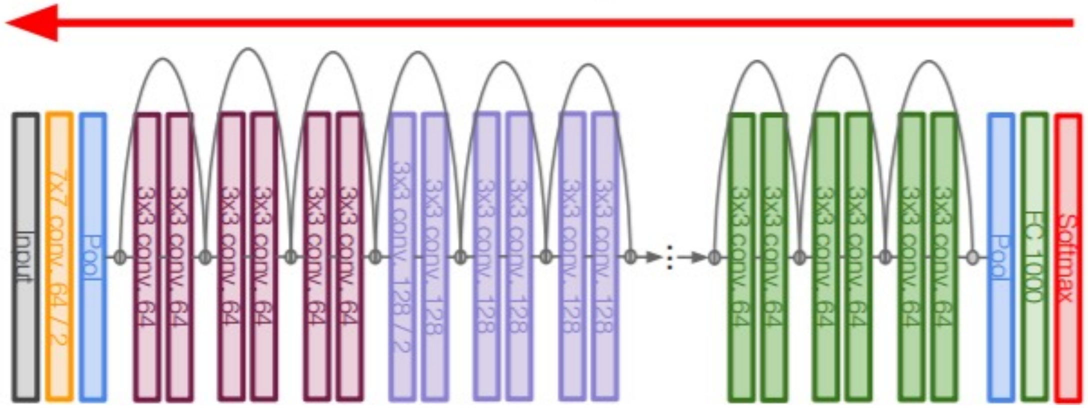
Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

Uninterrupted gradient flow!



Similar to ResNet!



In between:

Highway Networks

$$g = T(x, W_T)$$

$$y = g \odot H(x, W_H) + (1 - g) \odot x$$

Srivastava et al, "Highway Networks", ICML DL Workshop 2015

LSTM Achievements

- In 2013–2015, LSTMs started achieving state-of-the-art results
  - LSTMs have essentially replaced n-grams as language models for **speech**.
  - **Image captioning** and other multi-modal tasks which were very difficult with previous methods became feasible with LSTMs.
  - **Neural MT**: broken away from plateau of SMT, especially for grammaticality (partly because of characters/subwords), but not yet industry strength.
  - Many **traditional NLP tasks** work very well with LSTMs, but not necessarily the top performers: e.g., POS tagging and NER: Choi 2016.

Ann Copestake, Overview of LSTMs and word2vec, 2016.

GRUs

- Gated Recurrent Units (GRU) introduced by Cho et al. 2014

- **Update gate**

$$z_t = \sigma\left(W_z \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_z\right)$$

- **Reset gate**

$$r_t = \sigma\left(W_r \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_r\right)$$

- **Memory**

$$\hat{h}_t = \tanh\left(W_m \begin{bmatrix} r_t \circ h_{t-1} \\ x_t \end{bmatrix} + b_m\right)$$

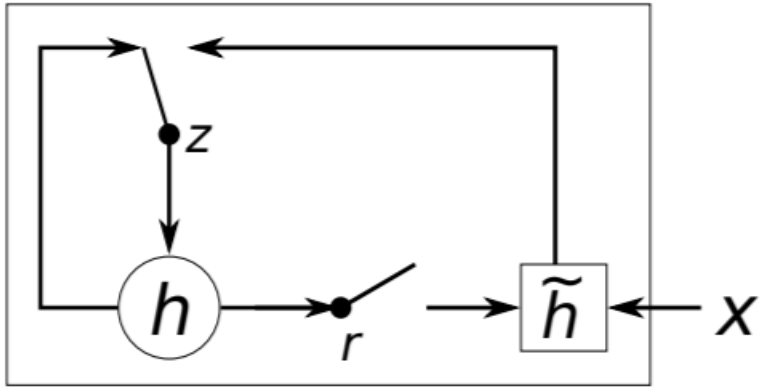
- **Final Memory**

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

If reset gate unit is ~0, then this ignores previous memory and only stores the new input

GRU intuition

- Units with long term dependencies have active update gates z
- Illustration:



This slide has been adopted from Socher lectures, cs224d, Stanford

GRU intuition

- If reset is close to 0, ignore previous hidden state

  →Allows model to drop information that is irrelevant in the future

- Update gate z controls how much of past state should matter now.
  - If z close to 1, then we can copy information in that unit through many time steps! Less vanishing gradient!

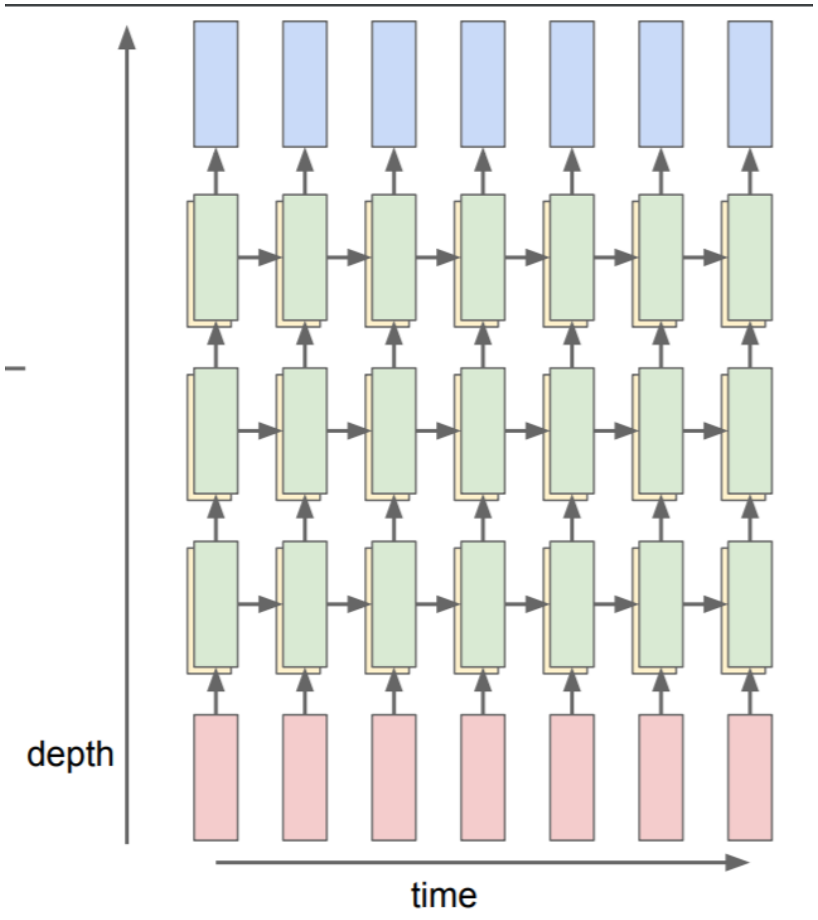
- Units with short-term dependencies often have reset gates very active

This slide has been adopted from Socher lectures, cs224d, Stanford

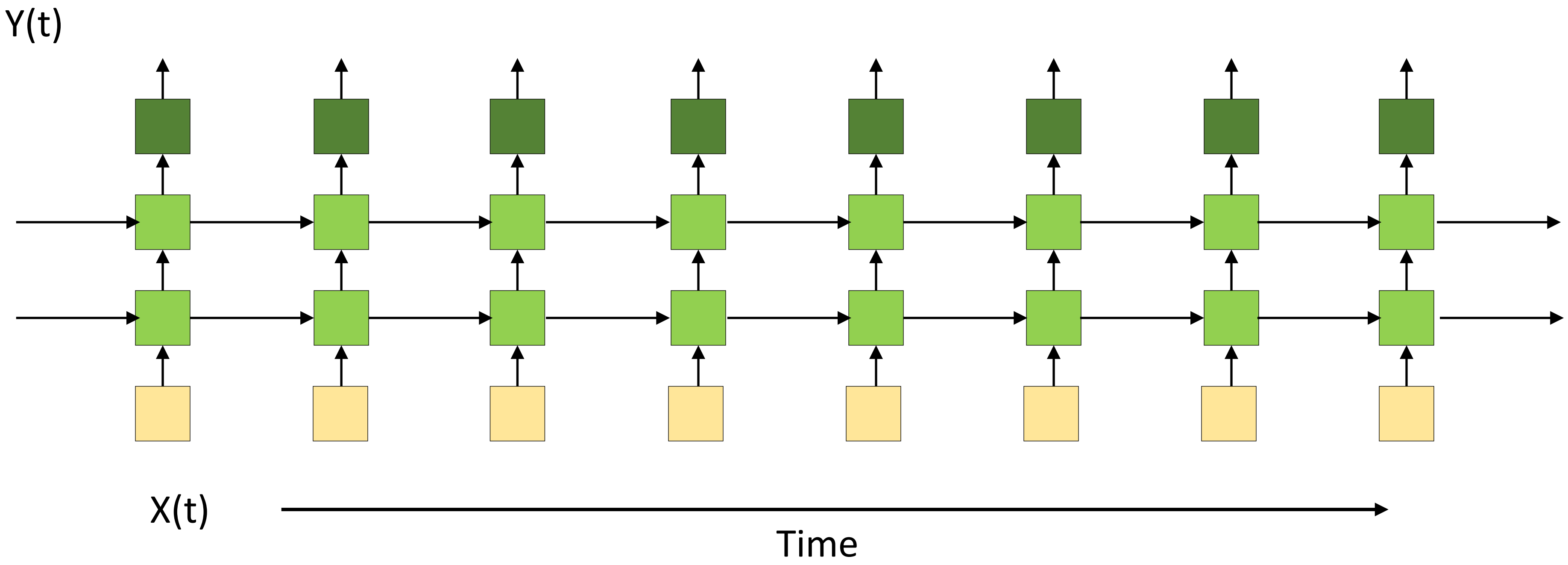
Multi-layer RNN

$$\boxed{h_t^l = \tanh W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}}$$

$h \in \mathbb{R}^n$. $\quad W^l \ [n \times 2n]$



Multi-layer LSTM architecture



- Each green box is now an entire LSTM or GRU unit
- Also keep in mind each box is an *array* of units

Story so far

- Recurrent networks are poor at memorization
  - Memory can explode or vanish depending on the weights and activation
- They also suffer from the vanishing gradient problem during training
  - Error at any time cannot affect parameter updates in the too-distant past
  - E.g. seeing a “close bracket” cannot affect its ability to predict an “open bracket” if it happened too long ago in the input
- LSTMs are an alternative formalism where memory is made more directly dependent on the input, rather than network parameters/structure
  - Through a memory structure with no weights or activations, but instead direct switching and “increment/decrement” from pattern recognizers
  - Do not suffer from a vanishing gradient problem but ***do* suffer from *exploding* gradient issue**

RNN: Summary

- RNNs allow a lot of flexibility in architecture design
- Vanilla RNNs are simple but don’t work very well
- Backward flow of gradients in RNN can explode or vanish.
  - Exploding is controlled with gradient clipping.
  - Vanishing is controlled with additive interactions (LSTM)
- Common to use LSTM or GRU: their additive interactions improve gradient flow

Sources:

- Sharif University of Technology, 40719 (DL Course), Spring 2025