CSE 465
Lecture 9

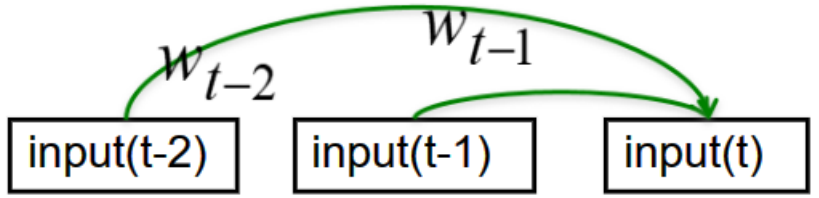
Sequence modelling with Neural Networks: RNN, LSTM and variants

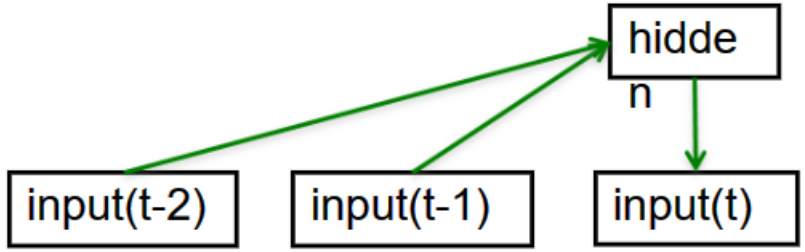
Modeling Sequences

- When applying machine learning to sequences, we often want to turn an input sequence into an output sequence that lives in a different domain
  - Turn a sequence of sound pressures into a sequence of word identities
- When there is no separate target sequence, we can get a training signal by trying to predict the next term in the input sequence
  - The target output sequence is the input sequence with an advance of 1 step
  - This seems much more natural than trying to predict one pixel in an image from the other pixels or one patch of an image from the rest of the image
  - For temporal sequences, there is a natural order for the predictions
- Predicting the next term in a sequence blurs the distinction between supervised and unsupervised learning
  - It uses methods designed for supervised learning, but it doesn't require a separate teaching signal

Memoryless models for sequences

- Autoregressive models
  - Predict the next term in a sequence from a fixed number of previous terms
- Feed-forward neural nets
  - These generalize autoregressive models by using one or more layers of non-linear hidden units





What is memory?

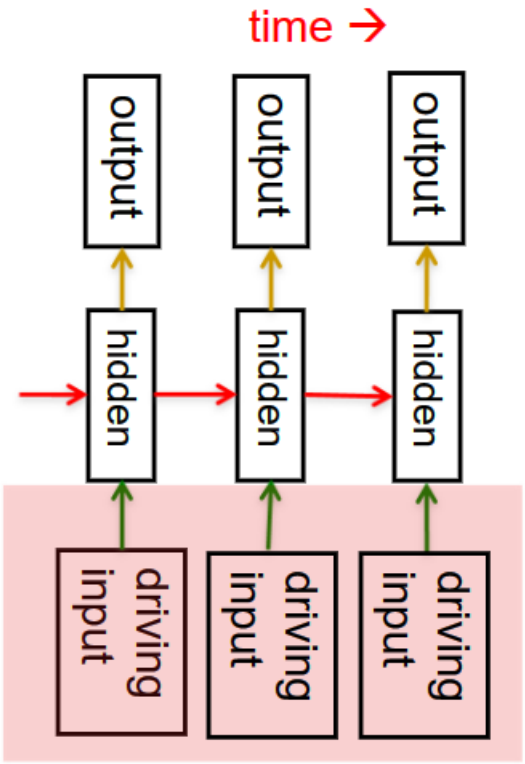
- Autoregressive models are memoryless, so they can't learn long-distance dependencies
- Recurrent neural networks (RNNs) are a kind of architecture that can remember things over time
- The Markov assumption:
  - $p(w_i|w_1, \ldots, w_{i-1}) = p(w_i|w_{i-3}, w_{i-2}, w_{i-1})$
    - Therefore, the model has a limited memory, i.e., it has no memory of anything before the last few words
    - But sometimes, long-distance context can be important

Adding memory to models

- If we give our generative model some hidden state, and if we give this hidden state its own internal dynamics, we get a much more interesting kind of model.
  - It can store information in its hidden state for a long time
  - If the dynamics are noisy and the way it generates outputs from its hidden state is noisy, we can never know its exact hidden state
  - The best we can do is to infer a probability distribution over the space of hidden state vectors
- This inference is only tractable for two types of hidden state model

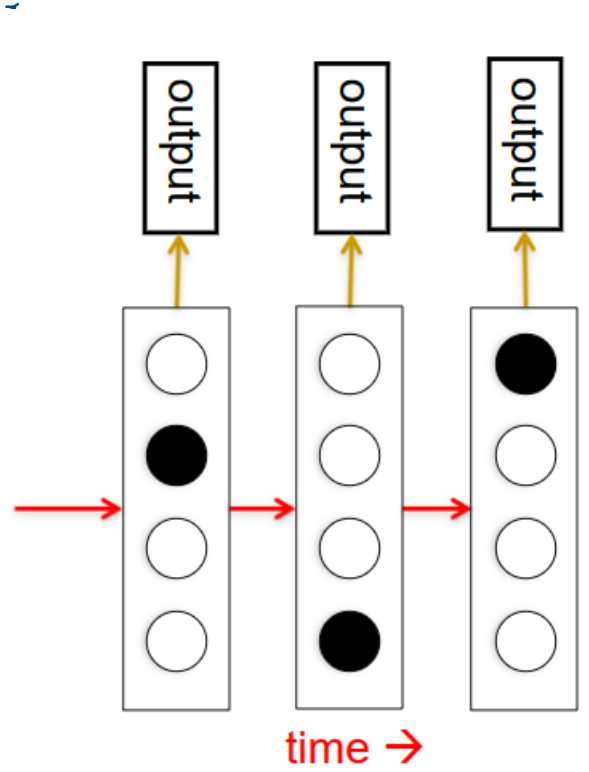
Linear Dynamical Systems

- These are generative models. They have a real-valued hidden state that cannot be observed directly.
  - The hidden state has linear dynamics with Gaussian noise and produces the observations using a linear model with Gaussian noise.
  - There may also be driving inputs.
- To predict the next output, we need to infer the hidden state.
  - A linearly transformed Gaussian is a Gaussian. So, the distribution over the hidden state given the data so far is Gaussian. It can be computed using "Kalman filtering".



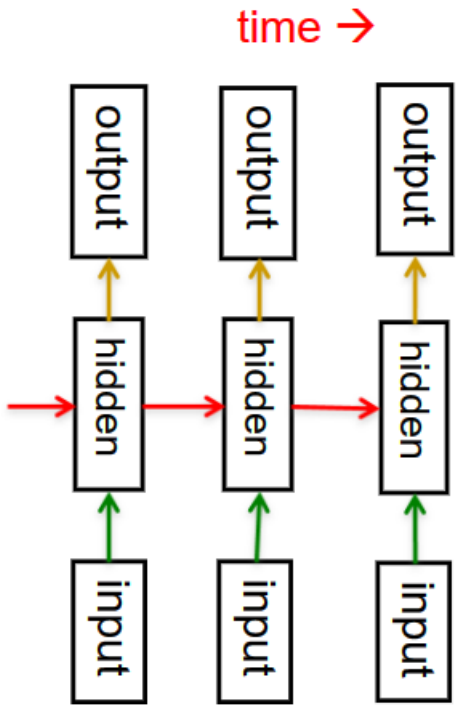
Hidden Markov Models

- Hidden Markov Models have a discrete one-of-N hidden state. Transitions between states are stochastic and controlled by a transition matrix. The outputs produced by a state are stochastic.
  - We cannot be sure which state produced a given output. So the state is “hidden”.
  - It is easy to represent a probability distribution across N states with N numbers.
- To predict the next output we need to infer the probability distribution over hidden states.
  - HMMs have efficient algorithms for inference and learning.



Recurrent neural networks

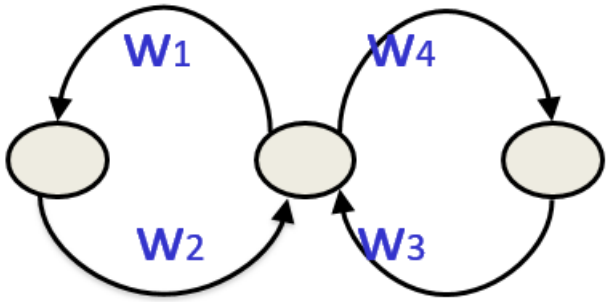
- RNNs are very powerful because they combine two properties:
  - Distributed hidden state that allows them to store a lot of information about the past efficiently.
  - Non-linear dynamics that allow them to update their hidden state in complicated ways.
- With enough neurons and time, RNNs can compute anything that can be computed by any computer.



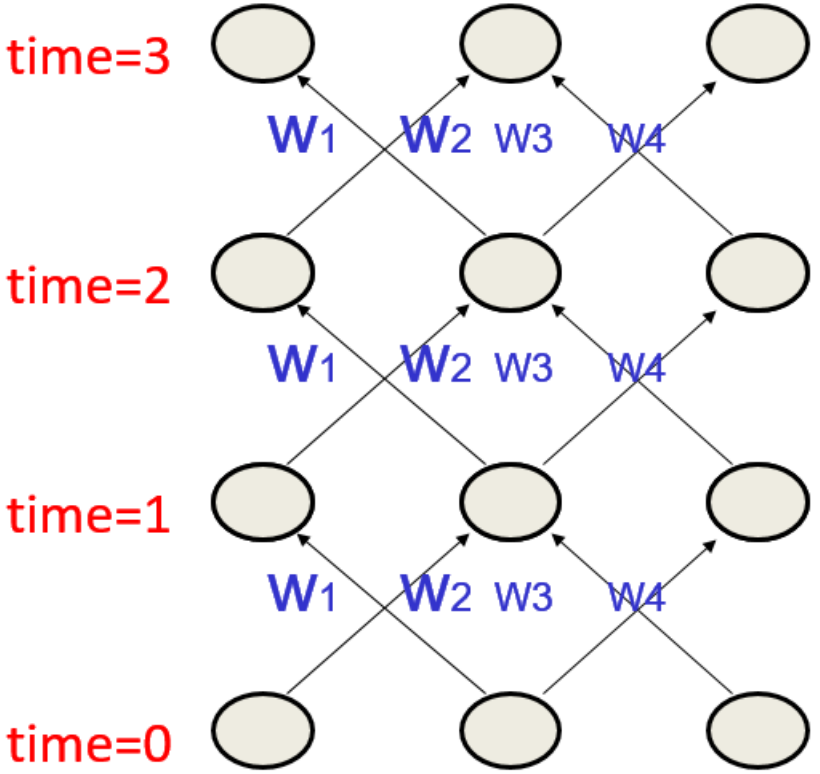
Recurrent neural networks

- What kinds of behavior can RNNs exhibit?
  - They can oscillate. Good for motor control?
  - They can settle to point attractors. Good for retrieving memories?
  - They can behave chaotically. Bad for information processing?
  - RNNs could potentially learn to implement many small programs that each capture a nugget of knowledge and run in parallel, interacting to produce complicated effects.
- But the computational power of RNNs makes them very hard to train.
  - For many years, we could not exploit the computational power of RNNs despite some heroic efforts

The equivalence between FF and RNN

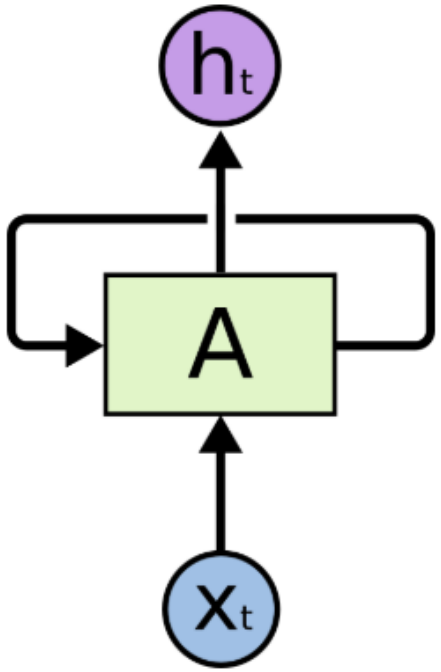


- Assume that there is a time delay of 1 in using each connection.
- The recurrent net is just a layered net that keeps reusing the same weights.

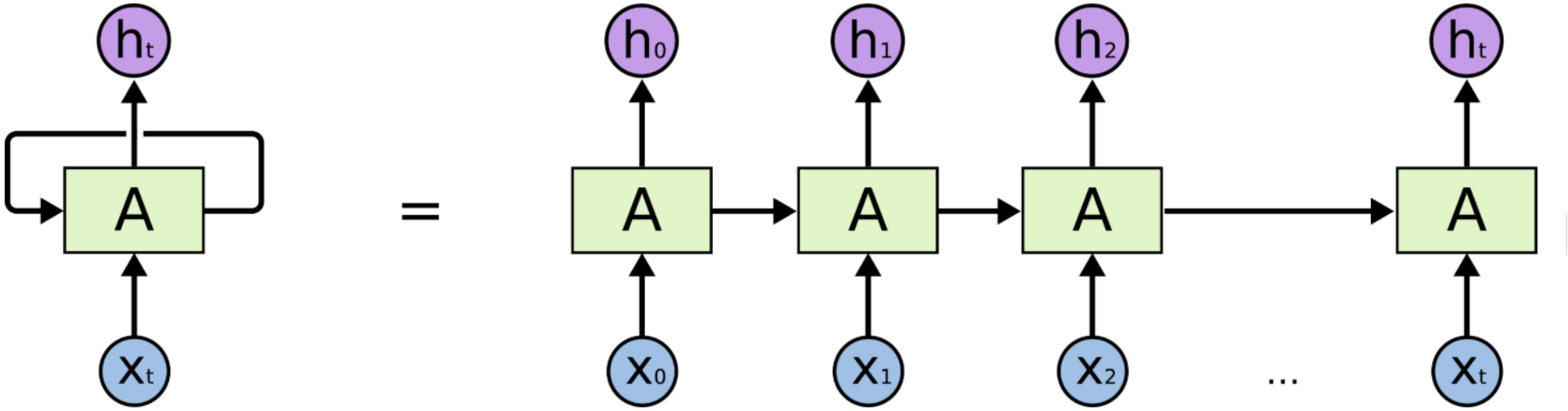


One RNN Unit

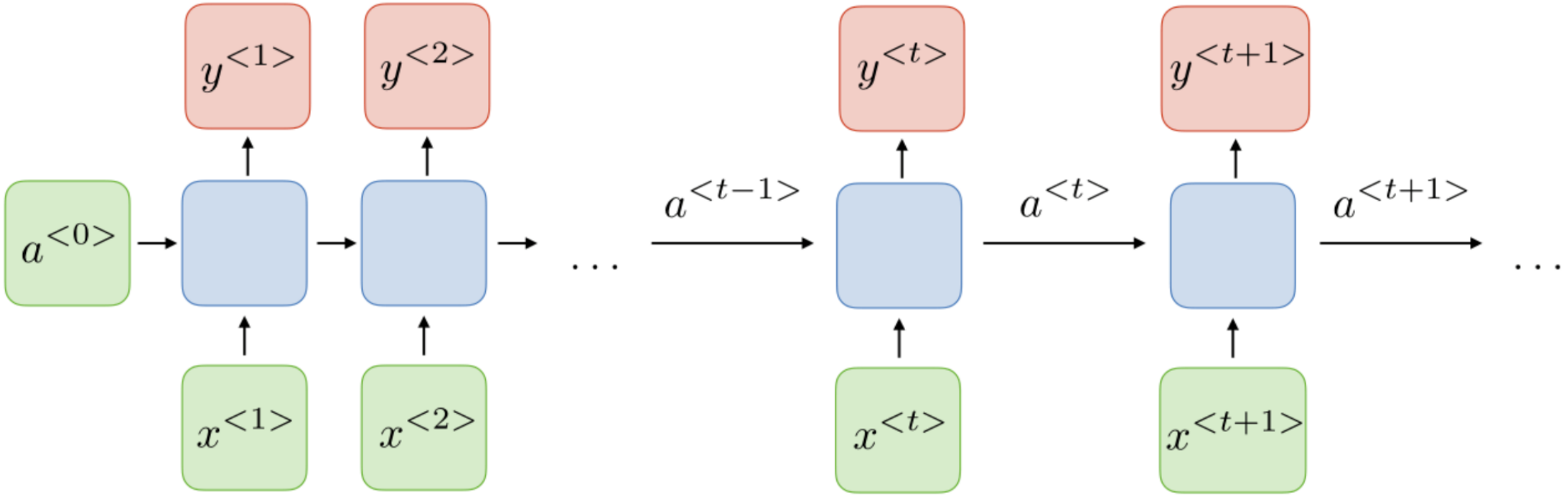
- And RNN
  - Update the hidden state in a deterministic nonlinear way
  - Then use the next symbol from the sequence as input to the same unit



How does it look?



With a little more details

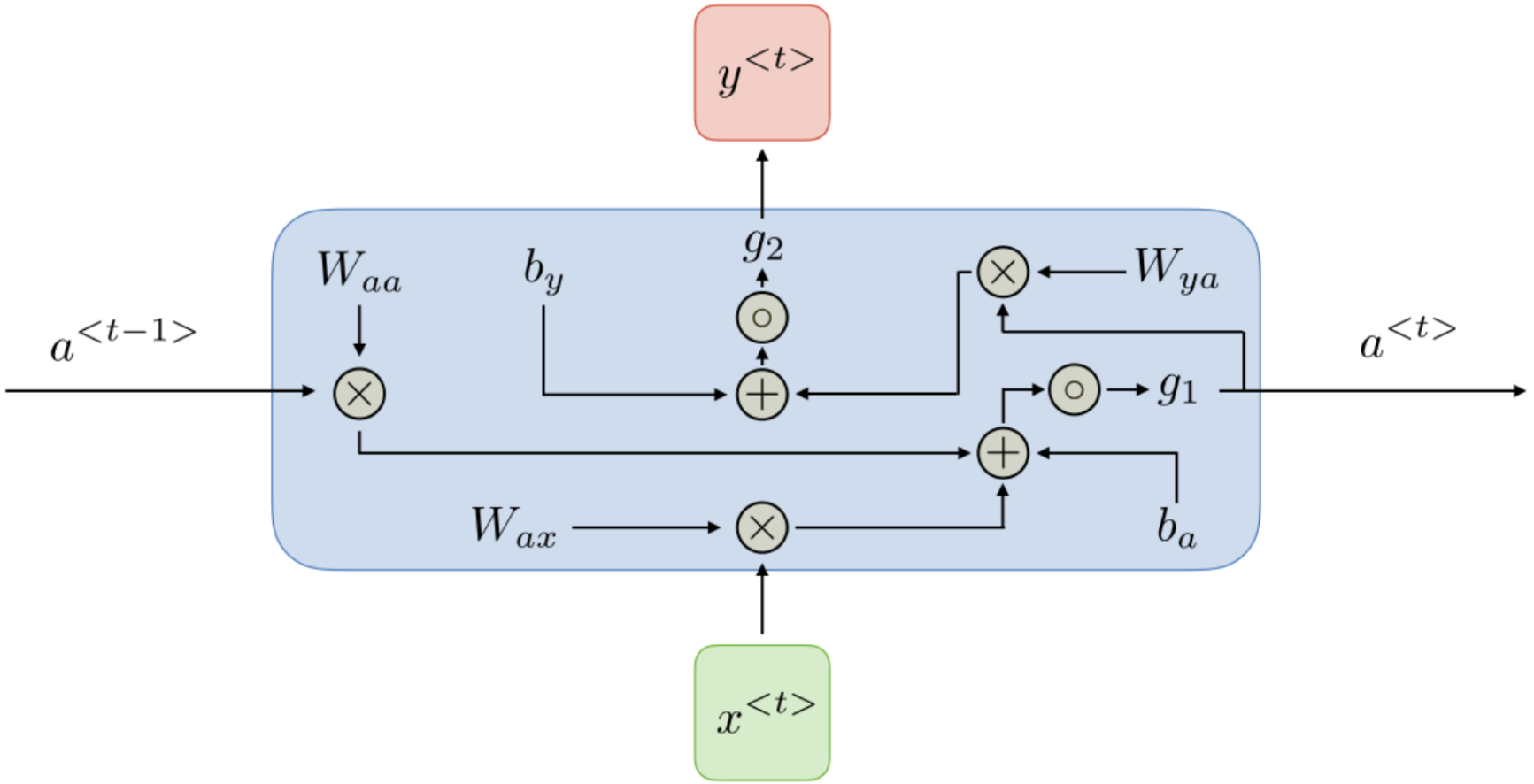


For each timestep t, the activation $a^{<t>}$ and the output $y^{<t>}$ are expressed as follows:

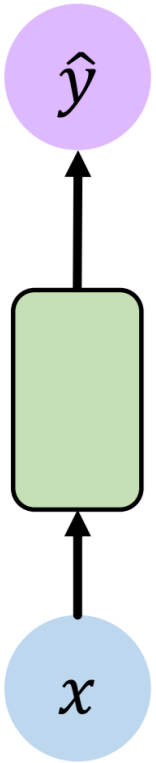
$$\boxed{a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)} \quad \text{and} \quad \boxed{y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)}$$

Detailed view of an RNN unit

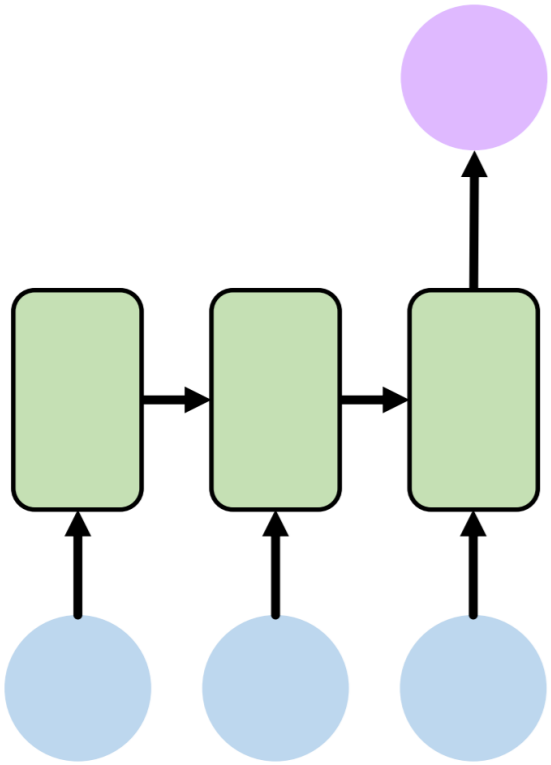
$W_{ax}, W_{aa}, W_{ya}, b_a, b_y$ are coefficients that are shared temporally and g_1, g_2 activation functions.



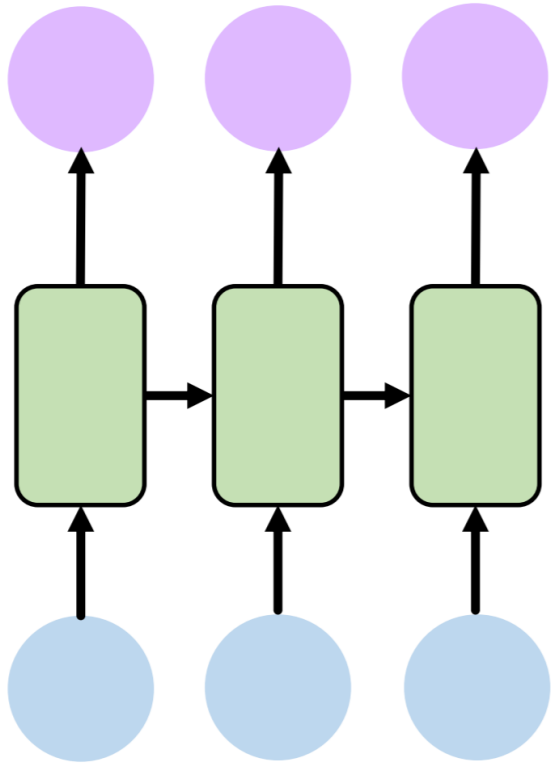
Types of RNN



One to One
"Vanilla" neural network

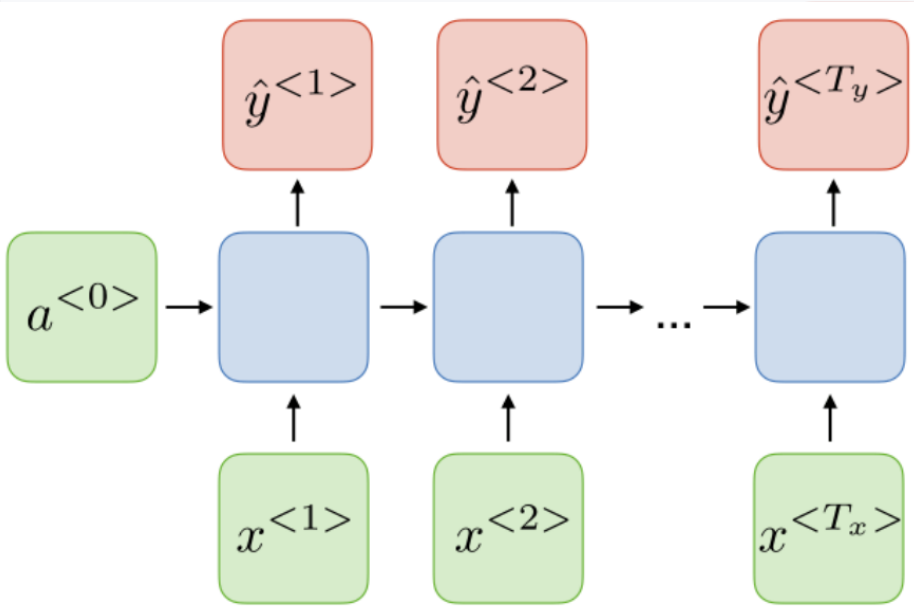


Many to One
Sentiment Classification

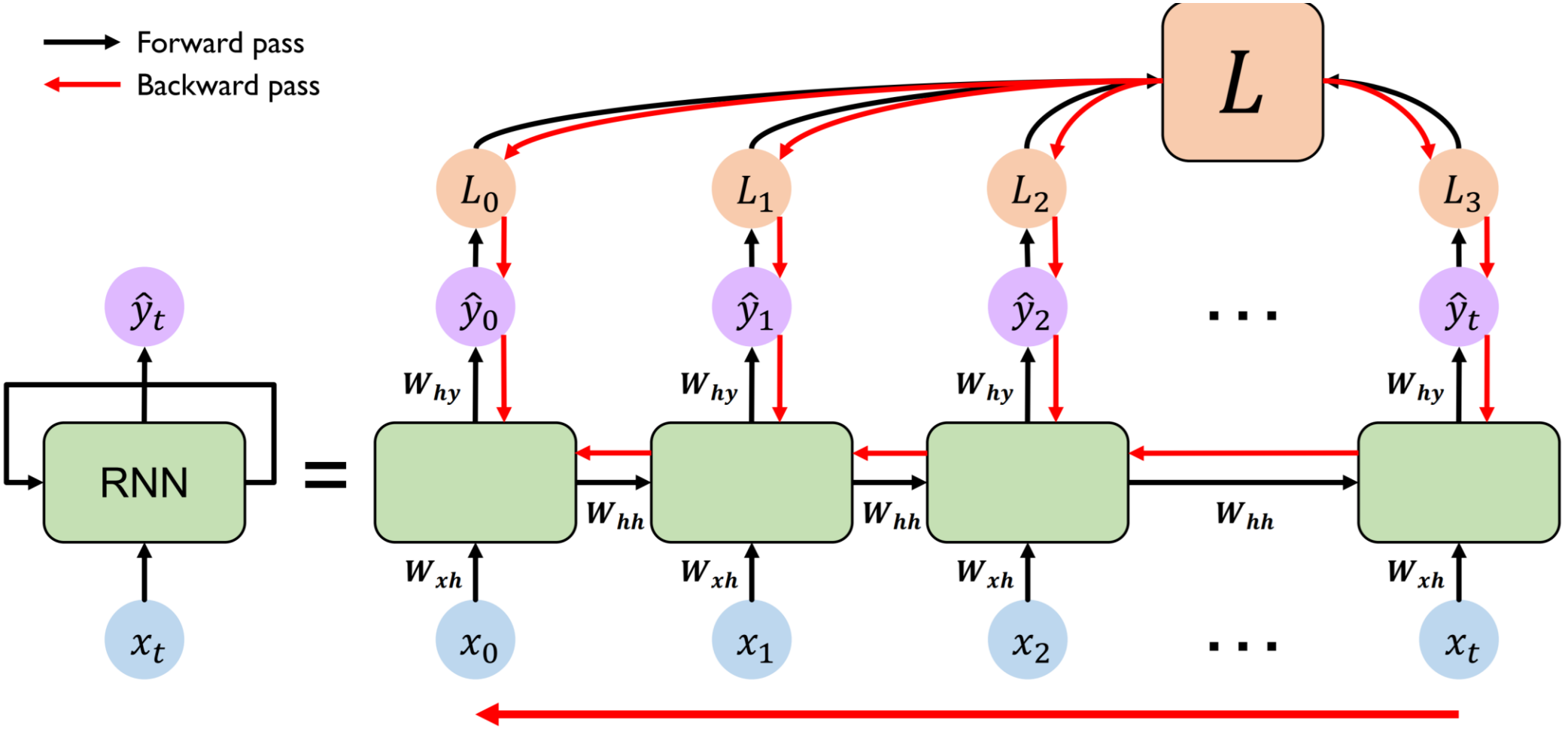


Many to Many
Music Generation

RNN Architecture for different applications

Many-to-many $T_x = T_y$		Name entity recognition

Backpropagation through time



Problem of long-term dependency

Why are vanishing gradients a problem?

Multiply many **small numbers** together

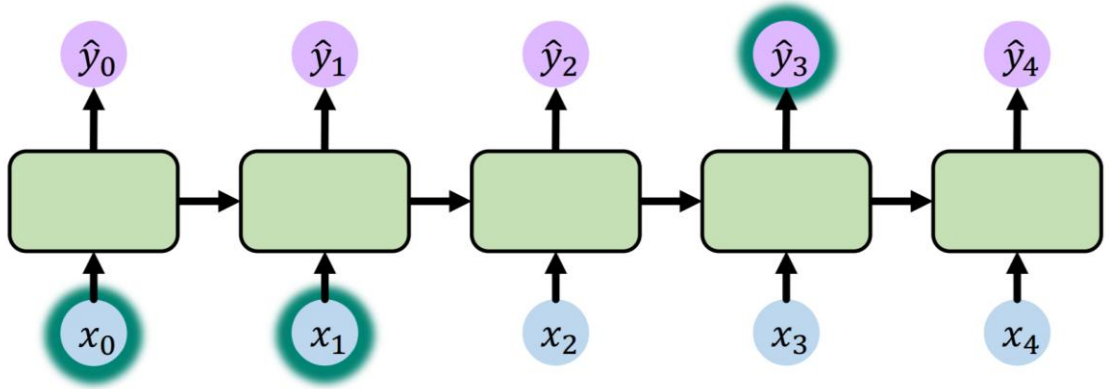
↓

Errors due to further back time steps have smaller and smaller gradients

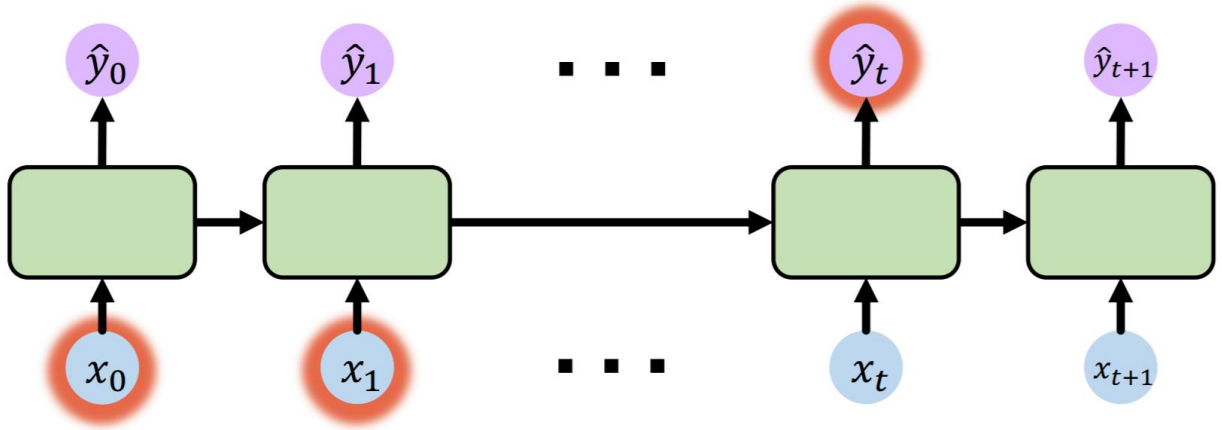
↓

Bias parameters to capture short-term dependencies

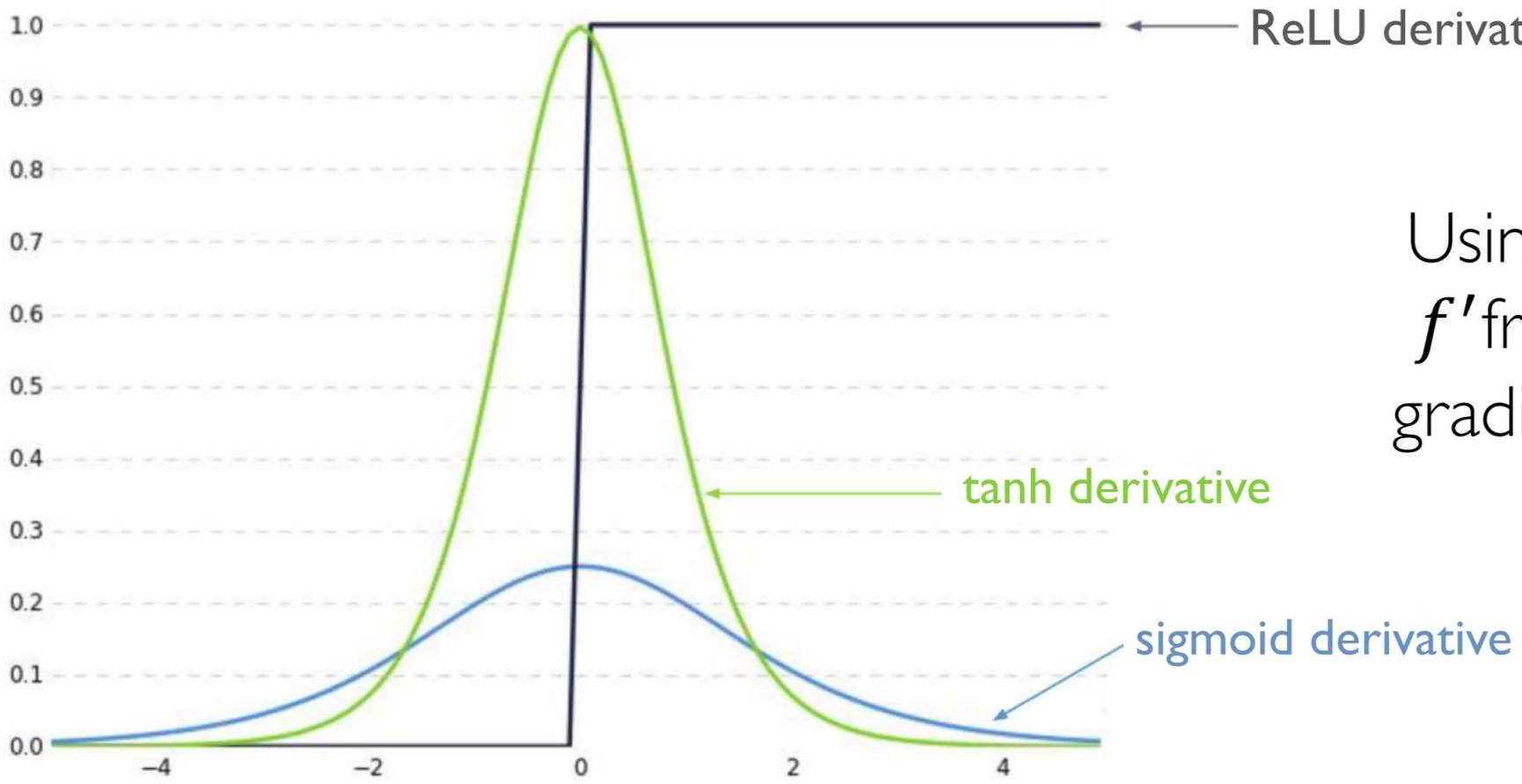
"The clouds are in the ___"



"I grew up in France, … and I I speak fluent___ "



Partial solution 1



Using ReLU prevents f' from shrinking the gradients when $x > 0$

Partial solution 2

Initialize **weights** to identity matrix

Initialize **biases** to zero

$$I_n = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

This helps prevent the weights from shrinking to zero.

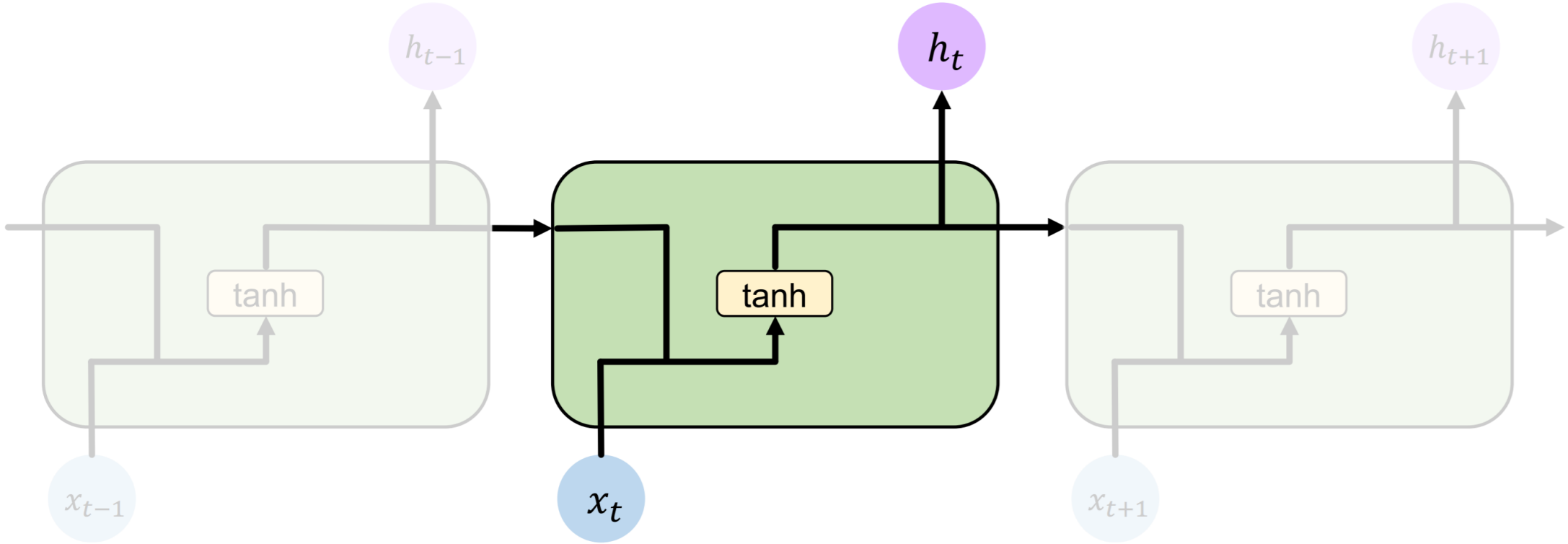
Main Solution – Different kind of units

Use a more complex recurrent unit with gates to control what information is passed through

Long Short Term Memory (**LSTMs**) networks rely on a gated cell to track information throughout many time steps

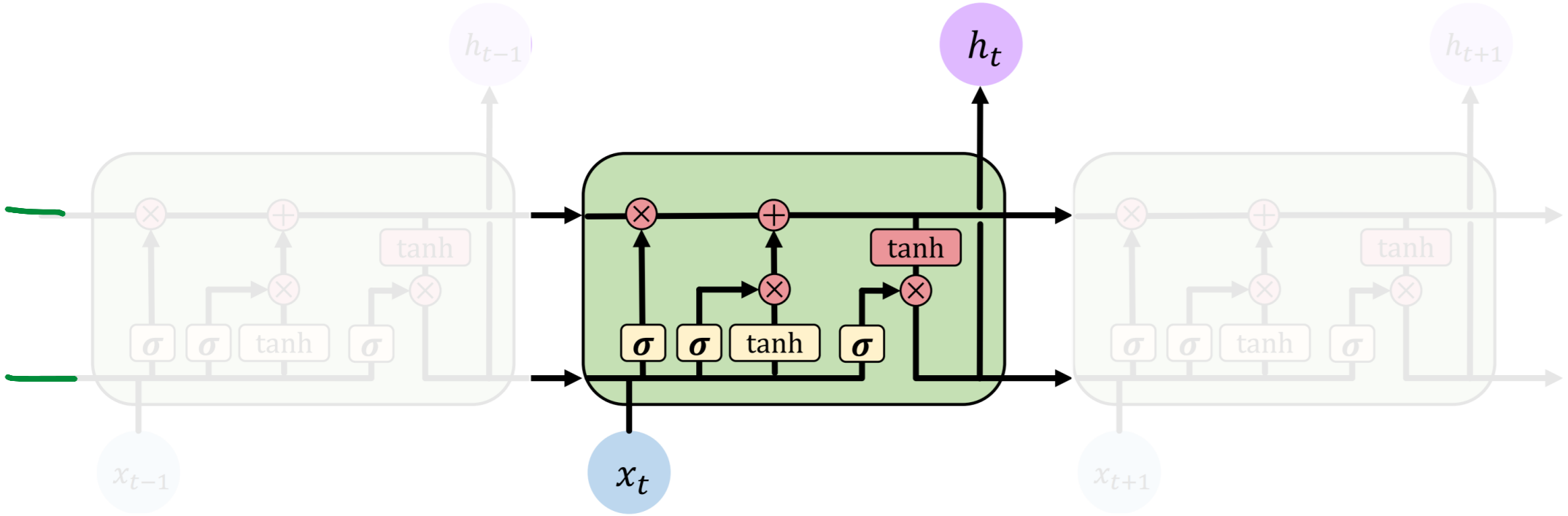
Standard RNN

In standard RNN repeating modules contain a simple computation unit

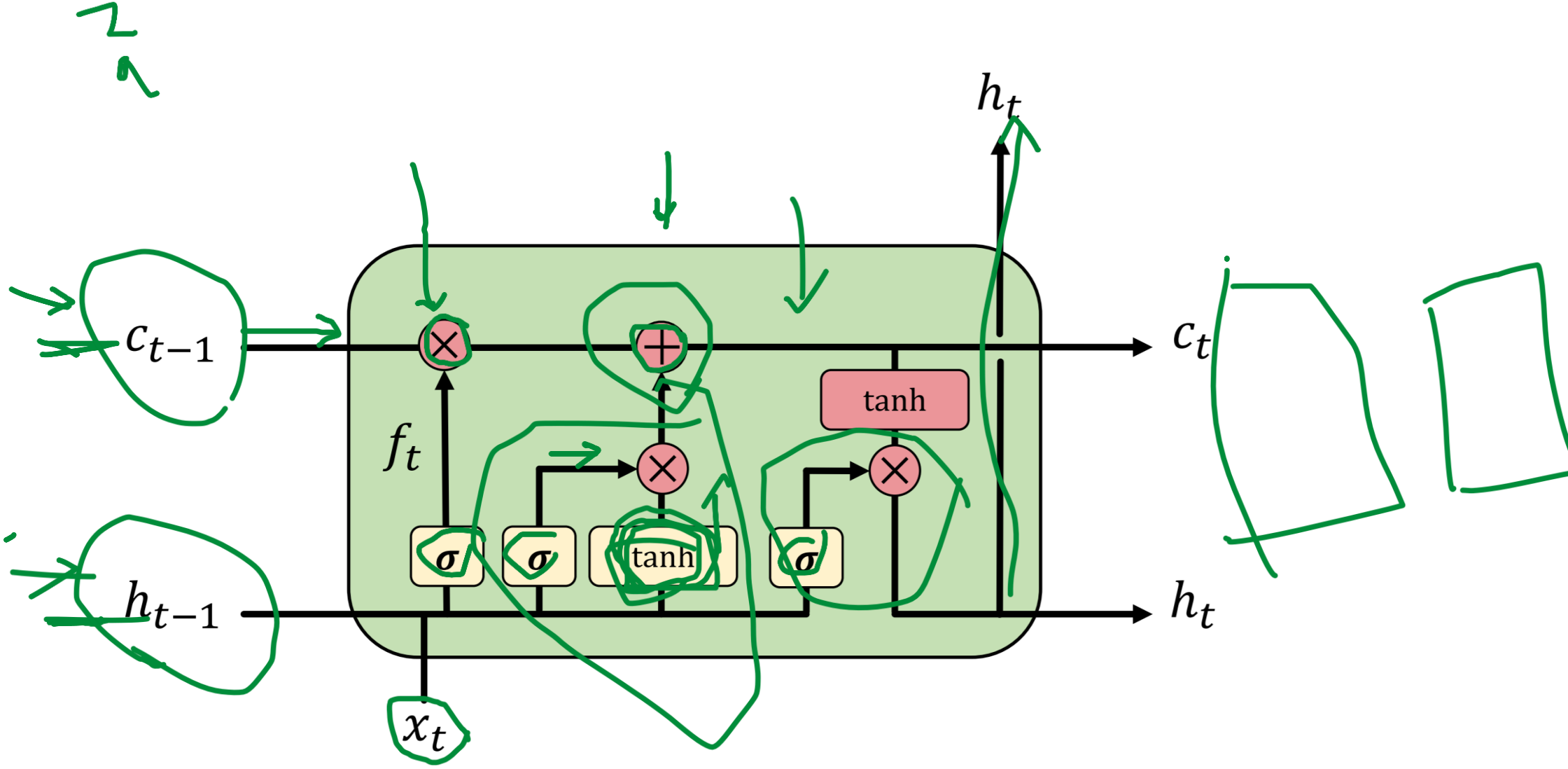


LSTM

- LSTM repeating modules contain interacting layers that control information flow
- LSTM cells can track information throughout many timesteps

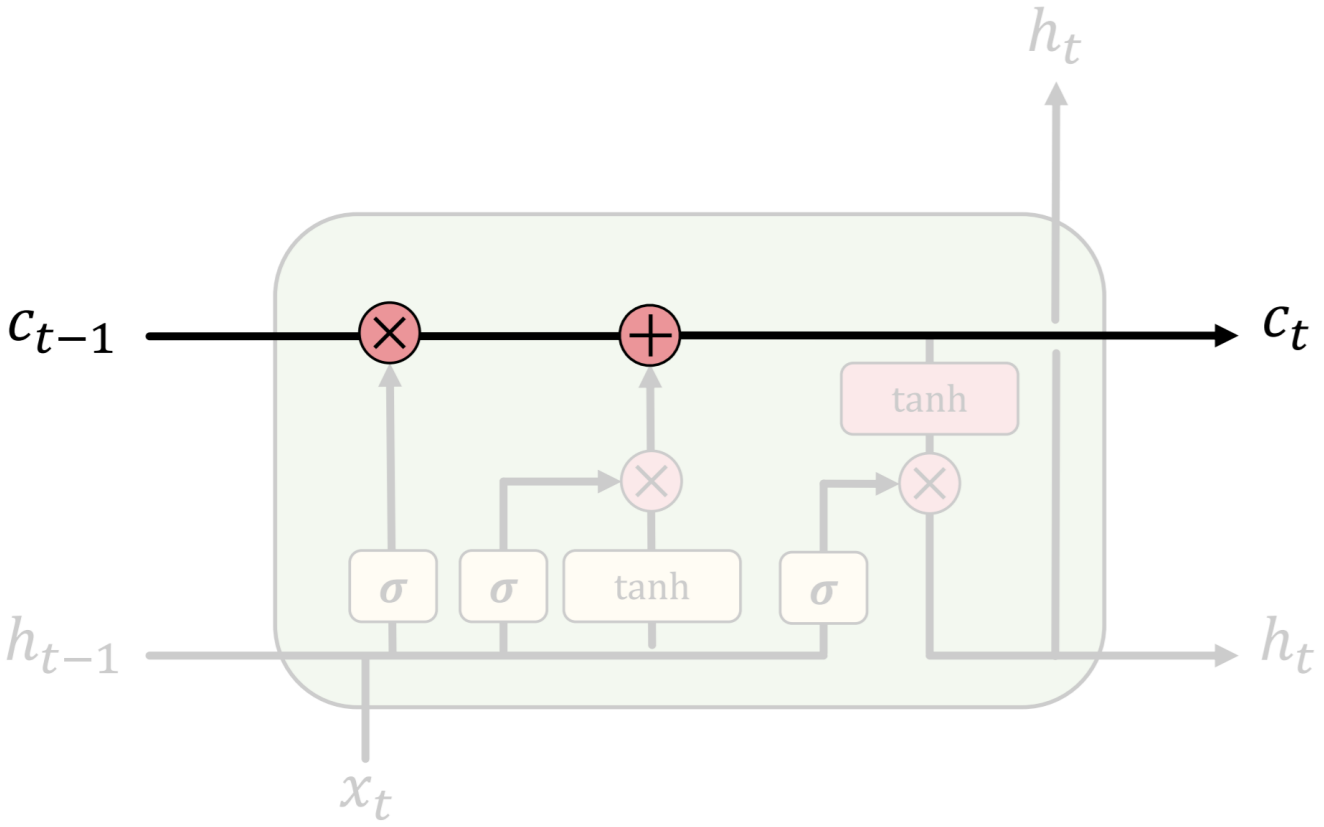


A complete LSTM unit



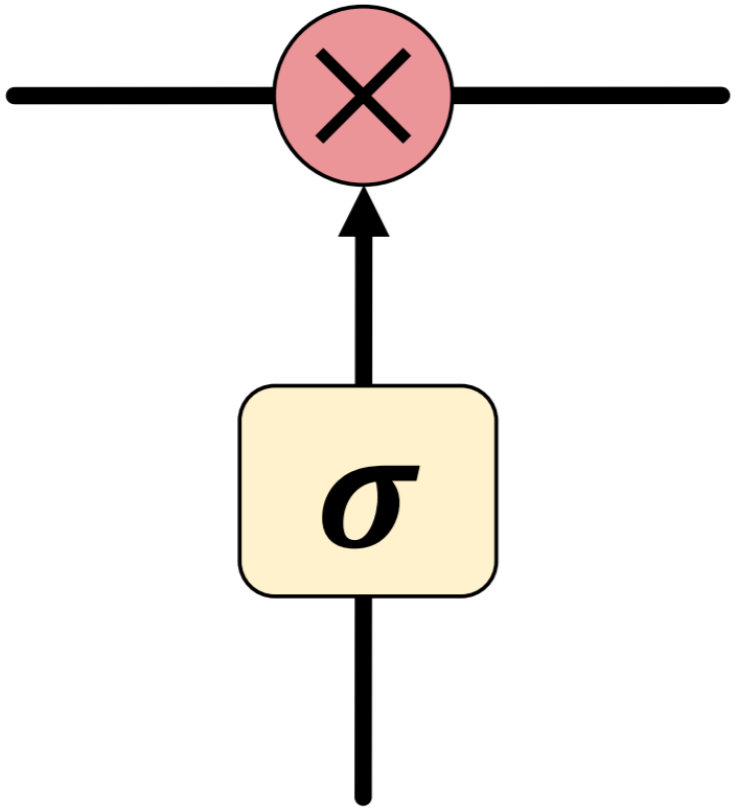
LSTM

- LSTMs maintain a cell state C_t where it's easy for information to flow



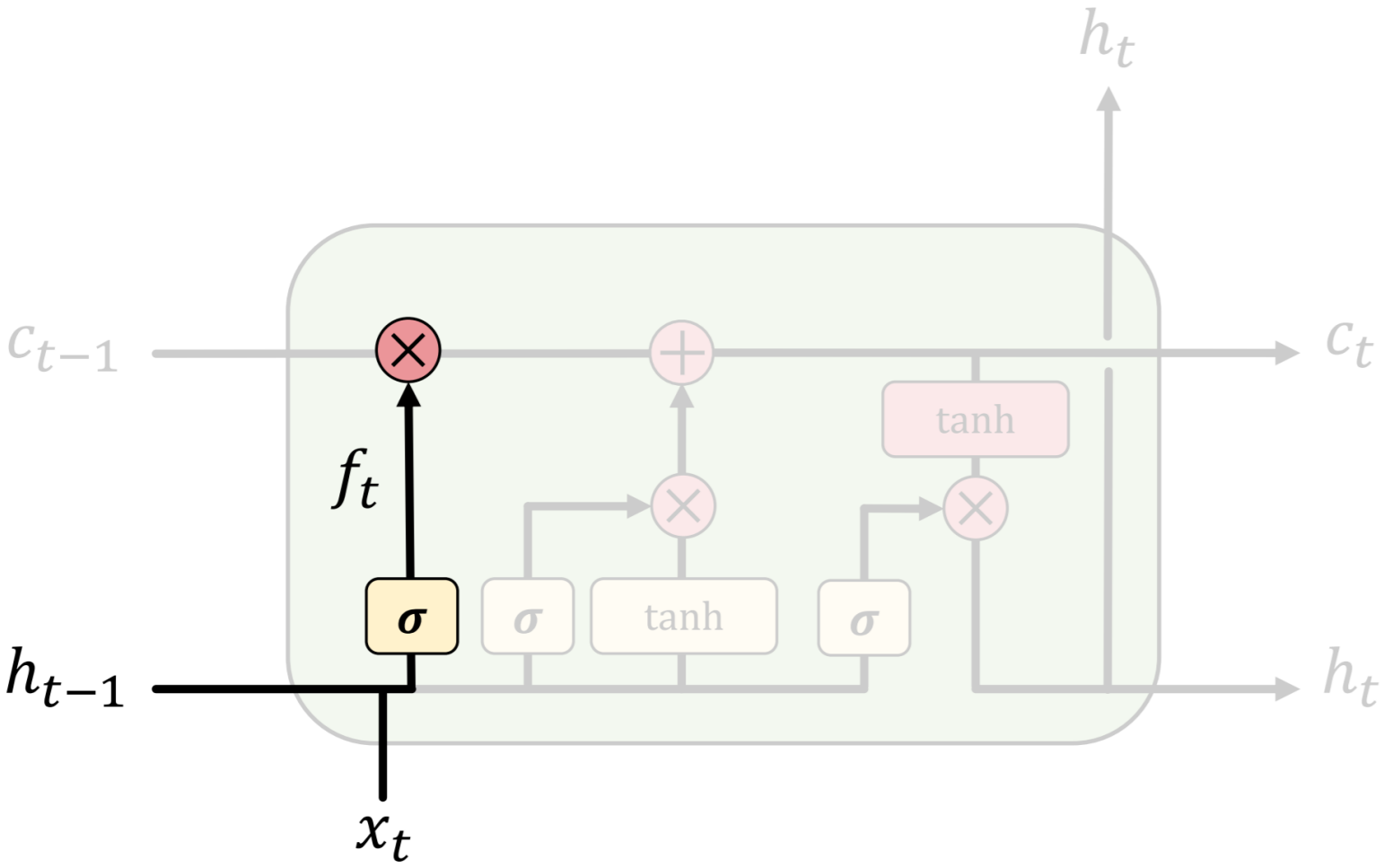
LSTM

- Information is added or removed to cell state through structures called gates
- Gates optionally let information through, via a sigmoid neural net layer and pointwise multiplication



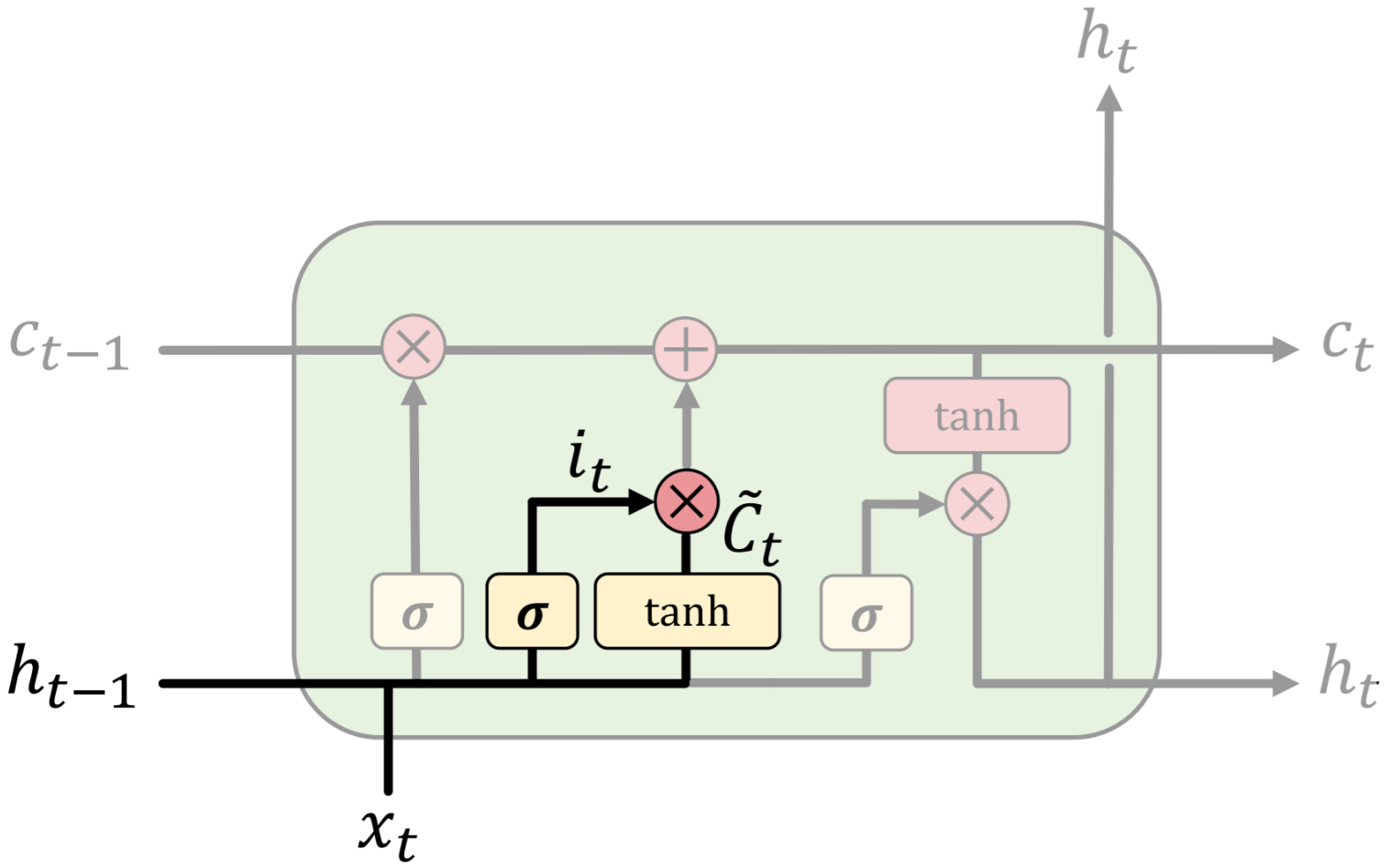
Forget gate

LSTMs forget irrelevant parts of the previous state: When we see a new subject we forget the old subject to pick the correct pronouns.



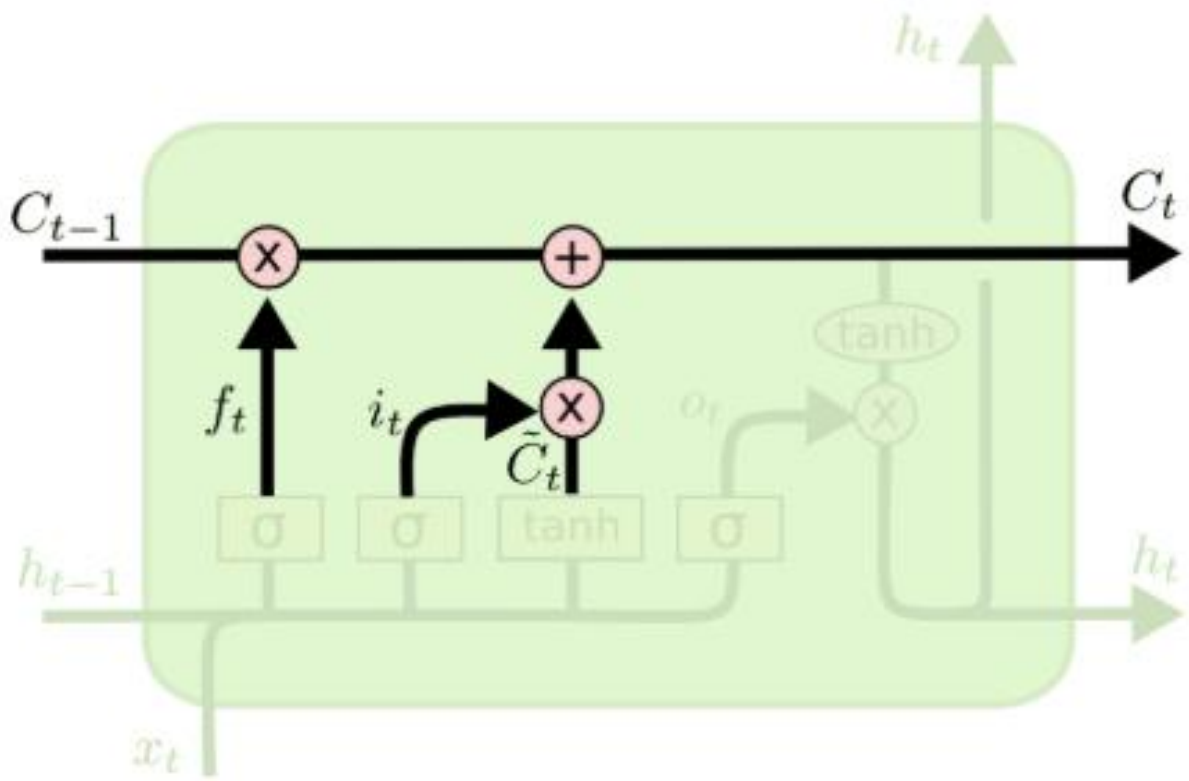
Update gate

LSTMs selectively update cell state values: maybe we want to add gender of the new subject



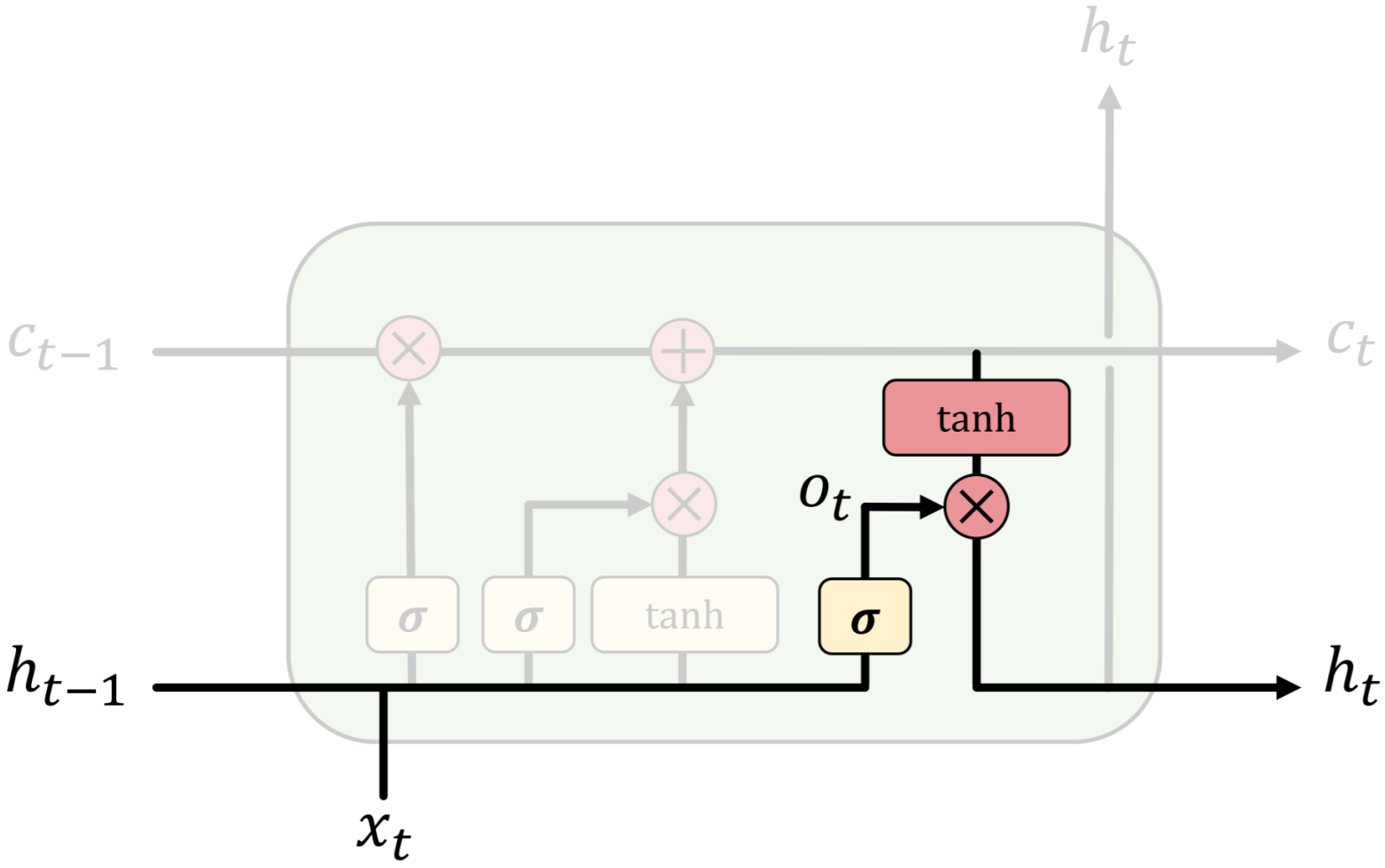
Total update

Update Ct: drop the information about the old subject's gender and add the new information



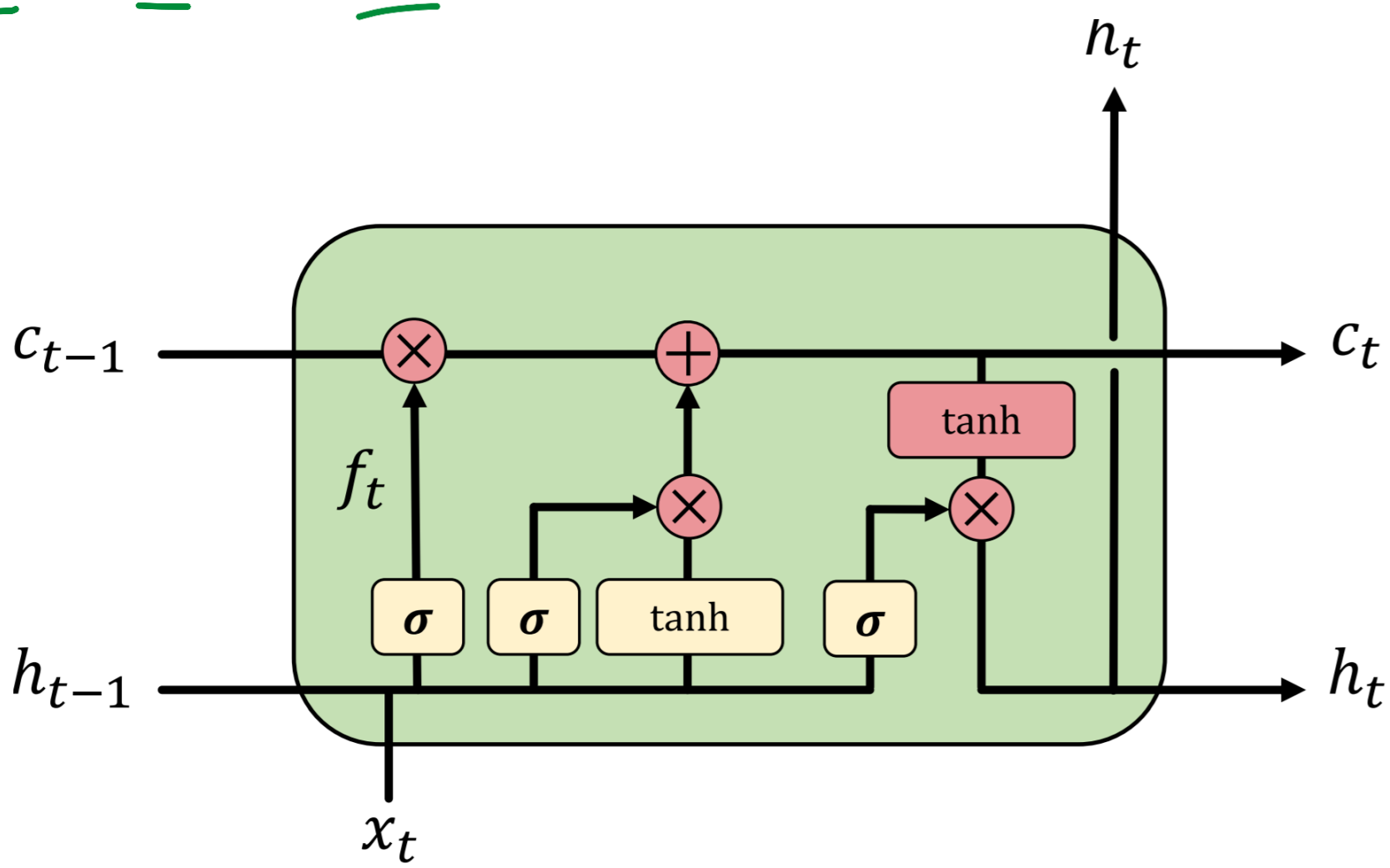
Output

- LSTMs use an output gate to output certain parts of the cell state: it might output whether the subject is singular or plural, so that we know what form a verb should be

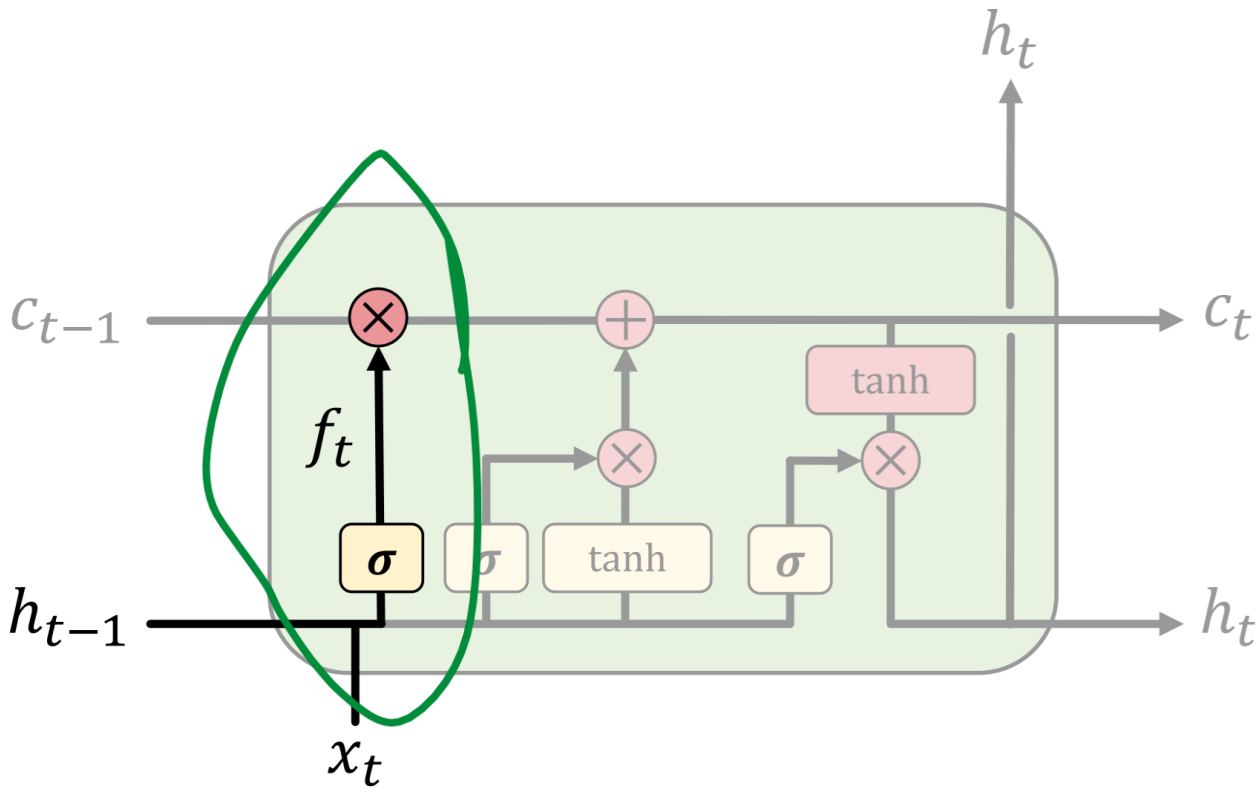


All gates together

- How do LSTMs work?
  - 1) Forget 2) Update 3) Output



LSTM: Forget irrelevant information

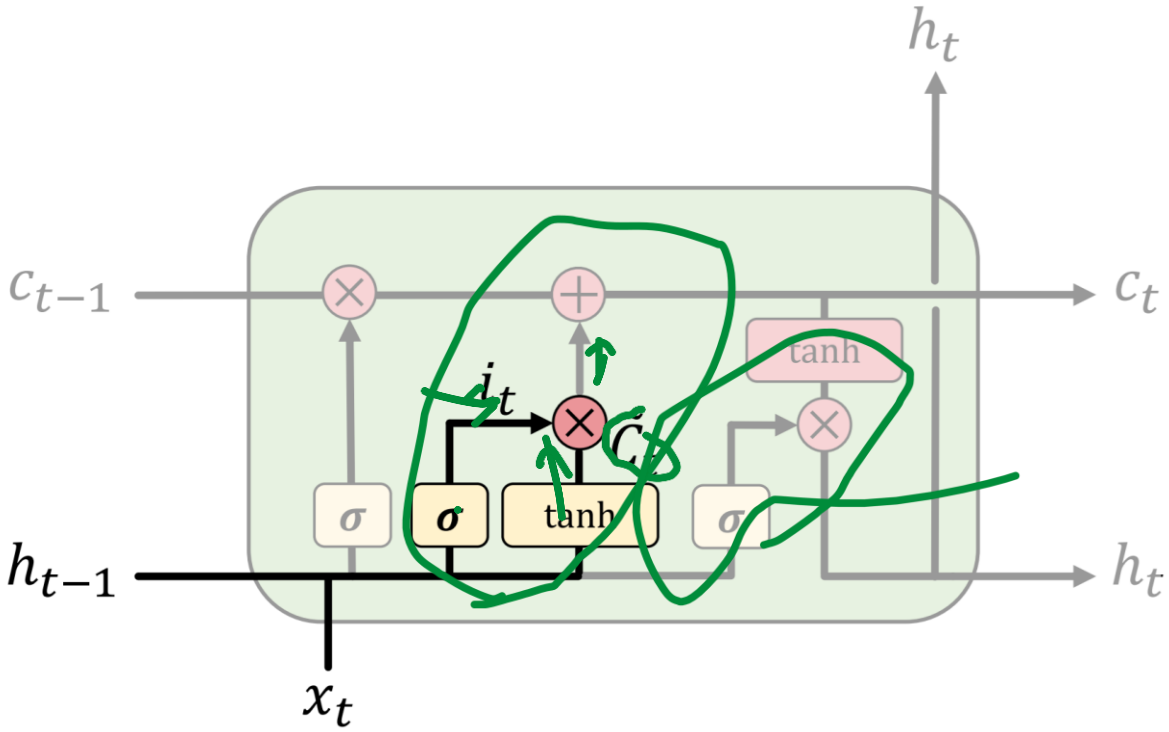


$$f_t = \sigma(\boldsymbol{W_i}[\, h_{t-1}, x_t \,] + b_f)$$

- Use previous cell output and input
- Sigmoid: value 0 and 1 – "completely forget" vs. "completely keep"

ex: Forget the gender pronoun of previous subject in sentence.

LSTM: Identify new information to store



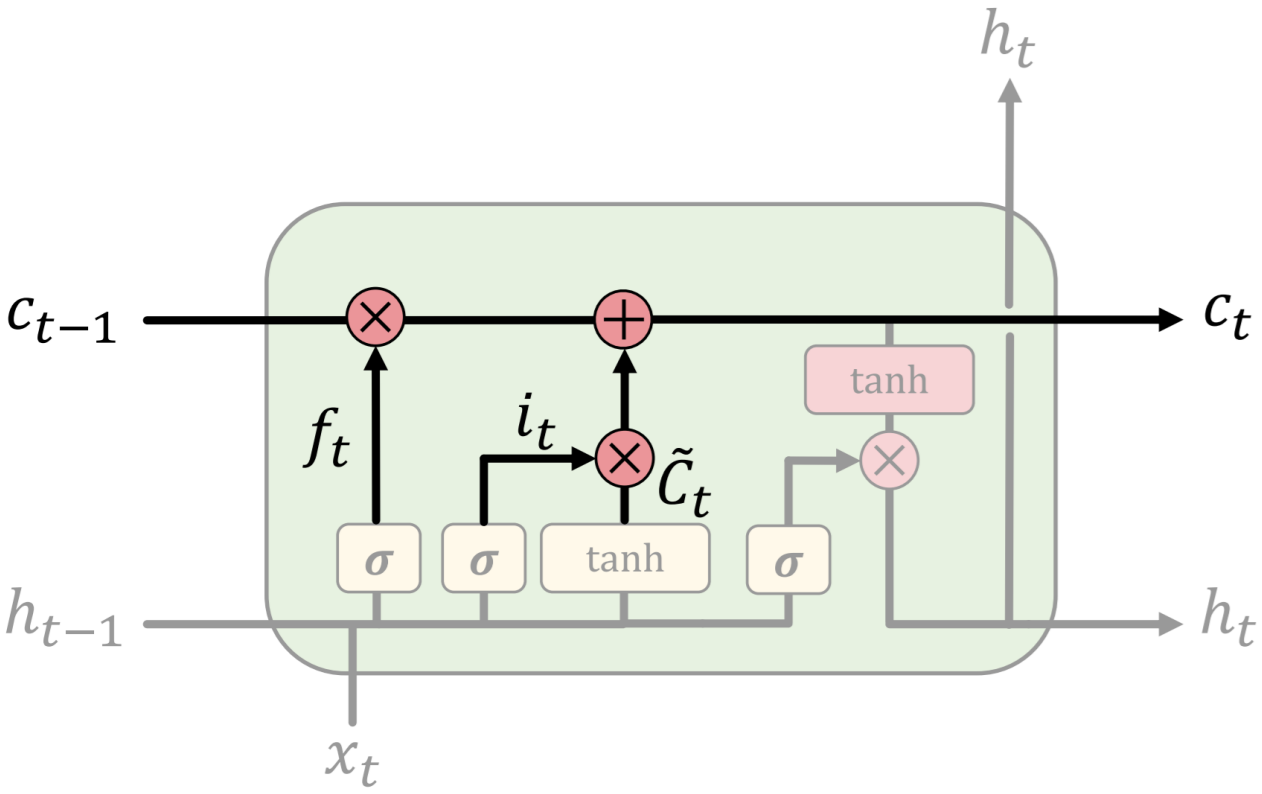
$$i_t = \sigma(\boldsymbol{W_i}[\, h_{t-1}, x_t \,] + b_i)$$

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

- Sigmoid layer: decide what values to update
- Tanh layer: generate new vector of "candidate values" that could be added to the state

ex: Add gender of new subject to replace that of old subject.

LSTM: Update new state

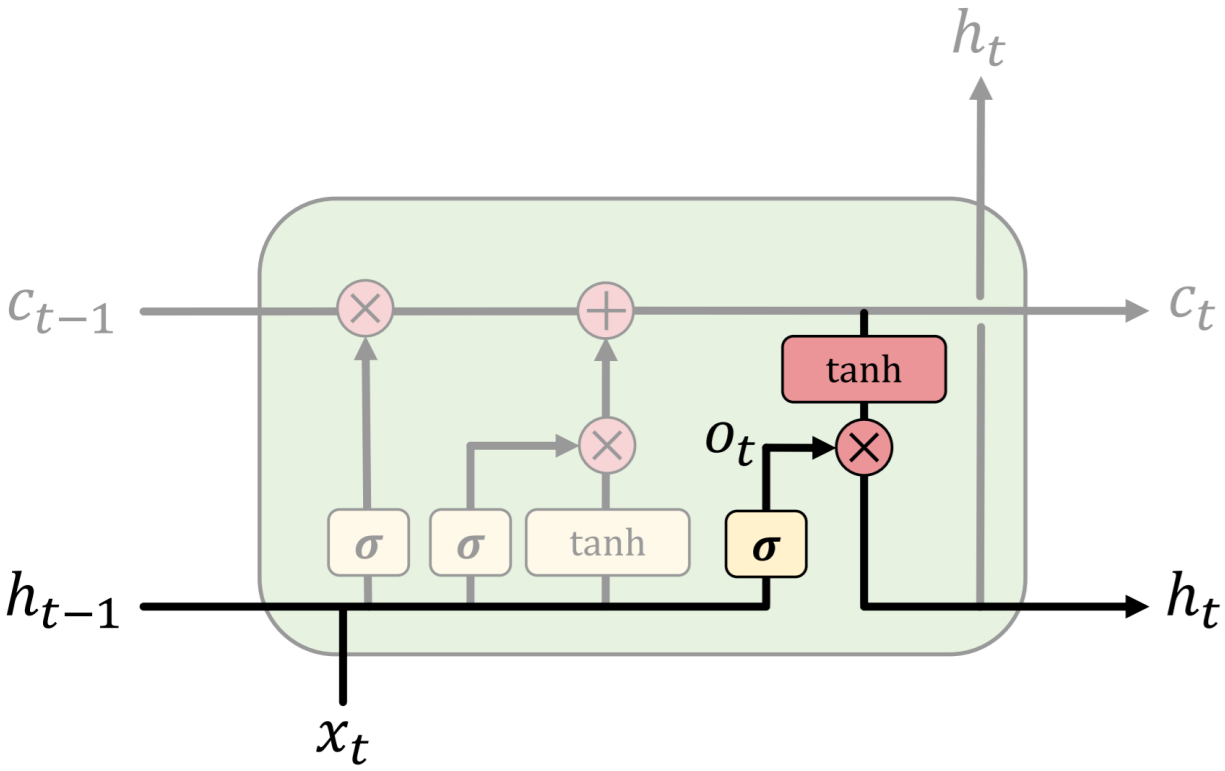


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

- Apply forget operation to previous internal cell state: $f_t * C_{t-1}$
- Add new candidate values, scaled by how much we decided to update: $i_t * \tilde{C}_t$

ex: Actually drop old information and add new information about subject's gender.

LSTM: output modified by the memory



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

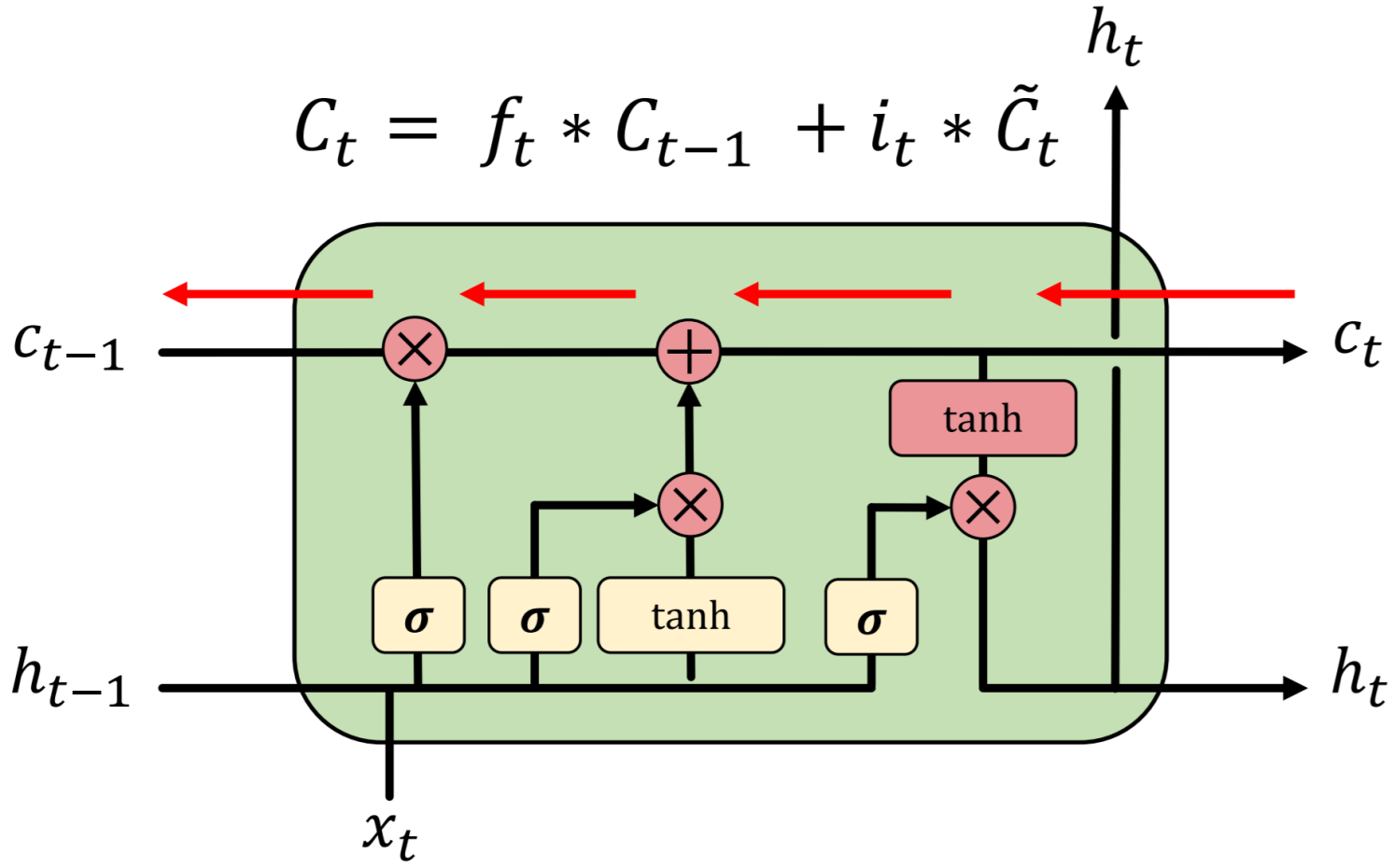
$$h_t = o_t * \tanh(C_t)$$

- Sigmoid layer: decide what parts of state to output
- Tanh layer: squash values between -1 and 1
- $o_t * \tanh(C_t)$: output filtered version of cell state

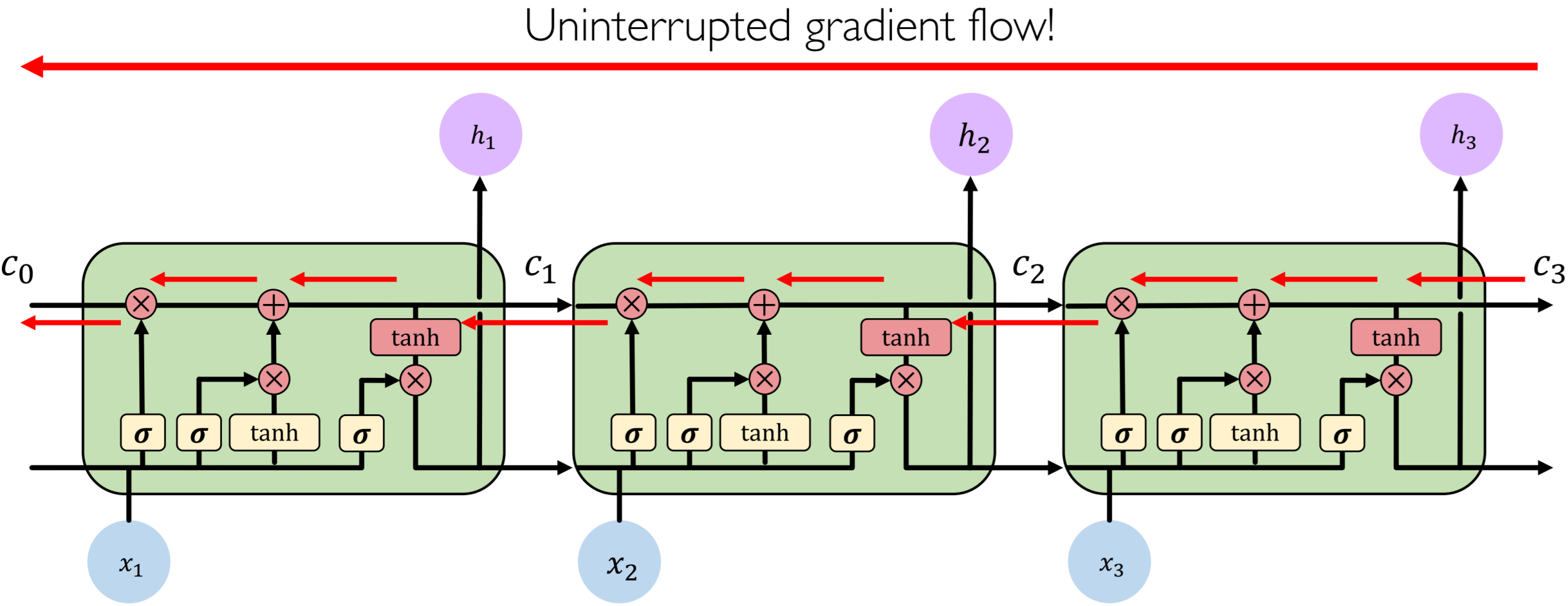
ex: Having seen a subject, may output information relating to a verb.

LSTM gradient flow

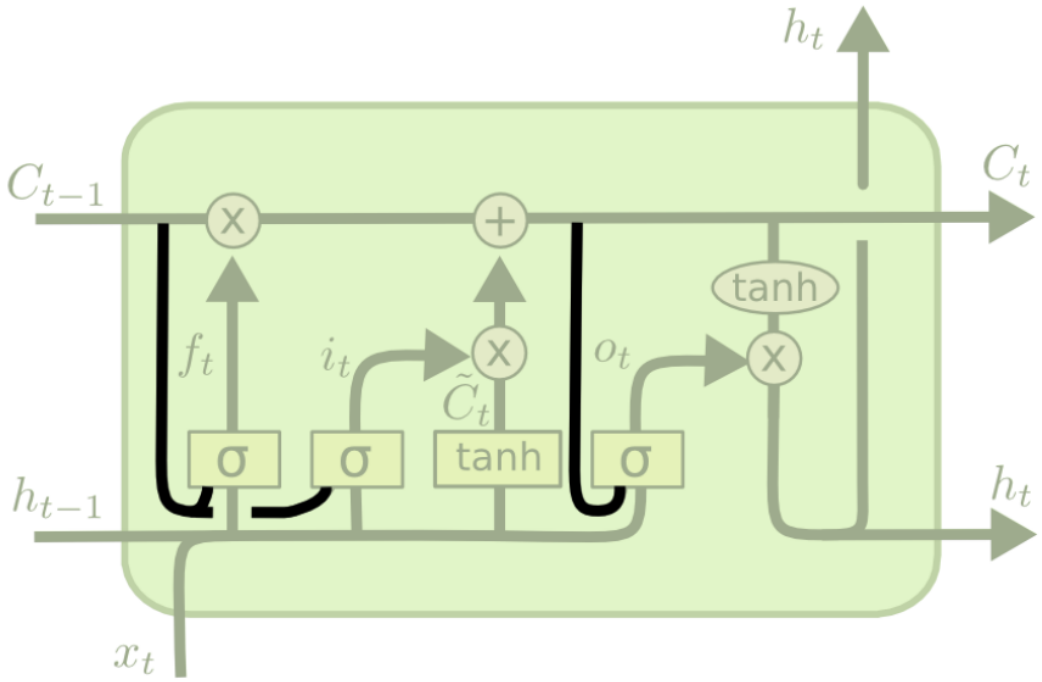
Backpropagation from C_t to C_{t-1} requires only elementwise multiplication!
No matrix multiplication → avoid vanishing gradient problem.



LSTM gradient flow



LSTM variant 1: Peephole connection

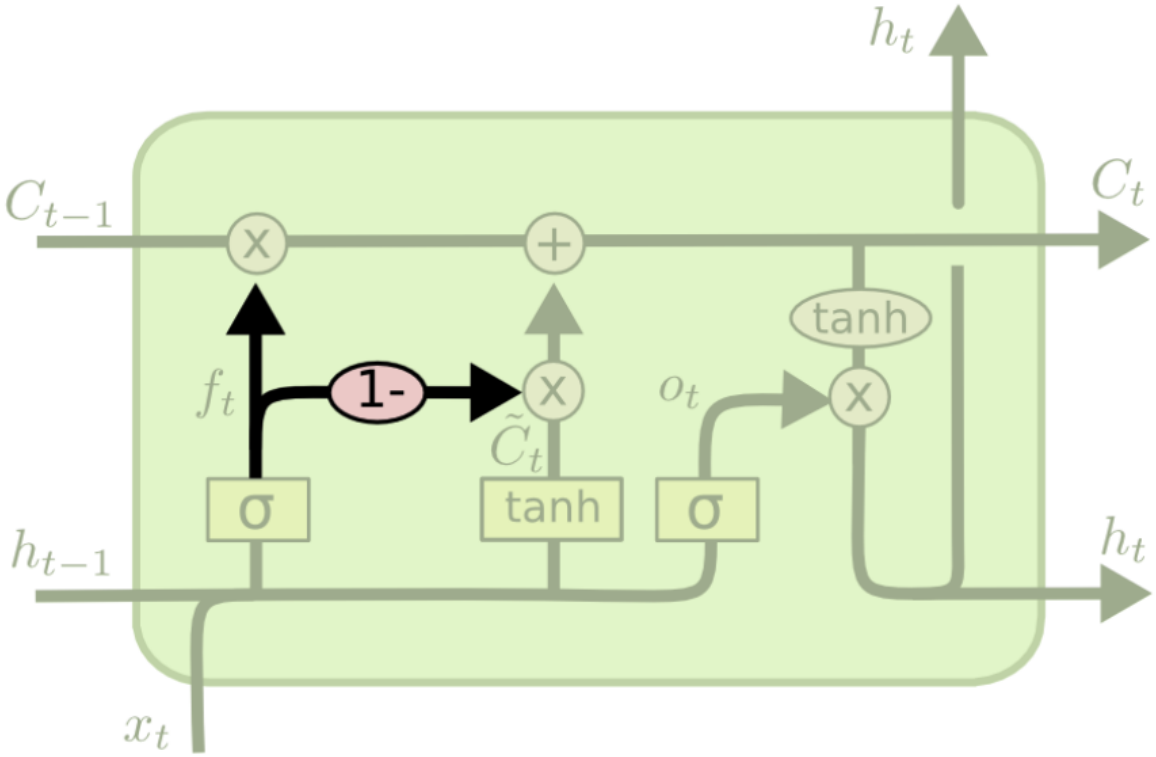


$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$

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

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

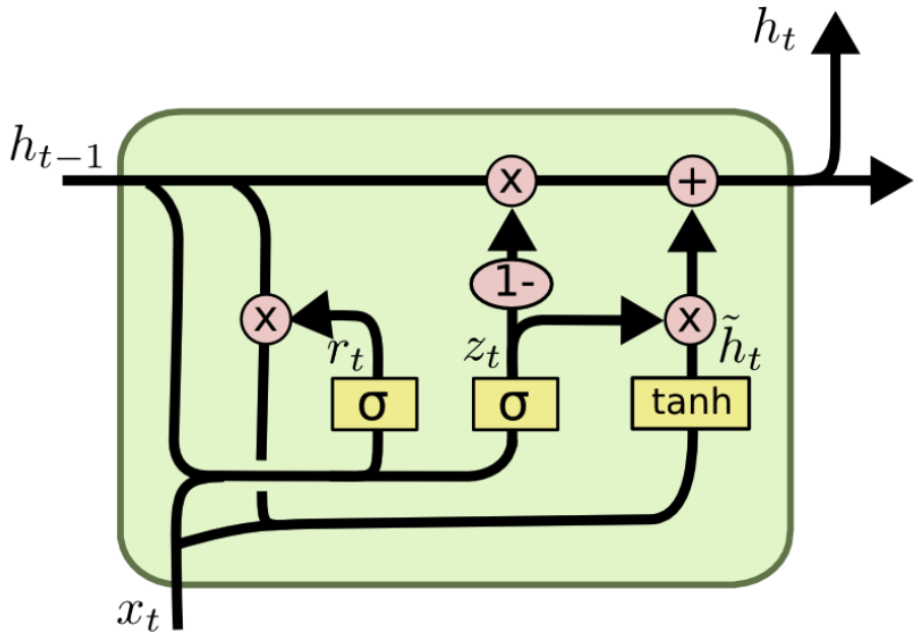
LSTM variant 2: couple forget and input



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

LSTM variant 3: GRU

- Combines the forget and input gates into a single “update gate”
- Merges the cell state and hidden state
- The resulting model is simpler than standard LSTM models, and has been growing increasingly popular



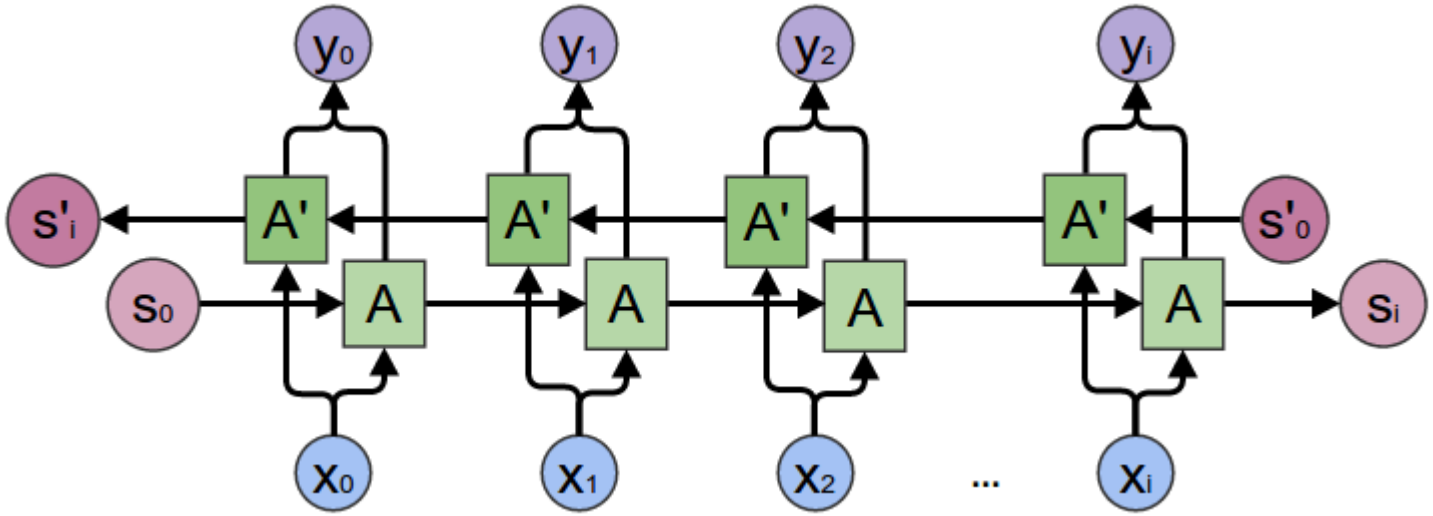
$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

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

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

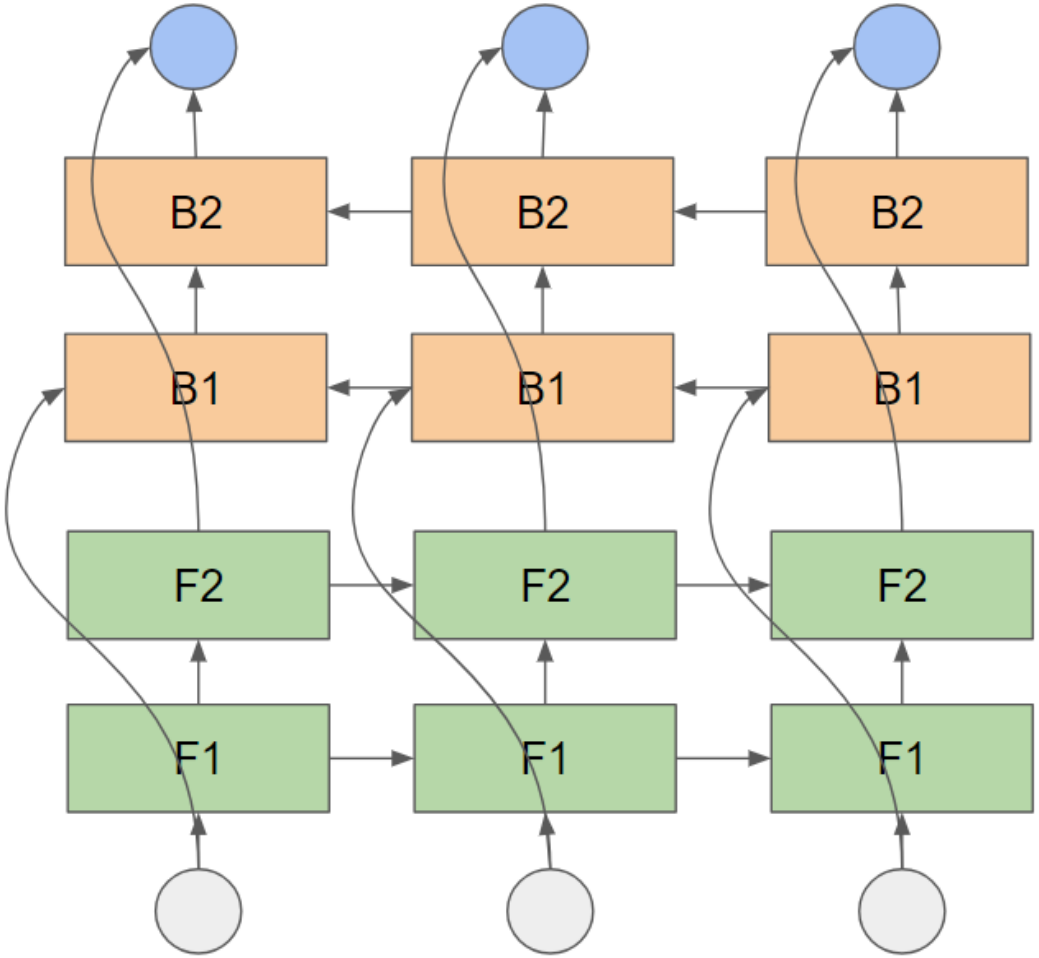
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Bidirectional RNN

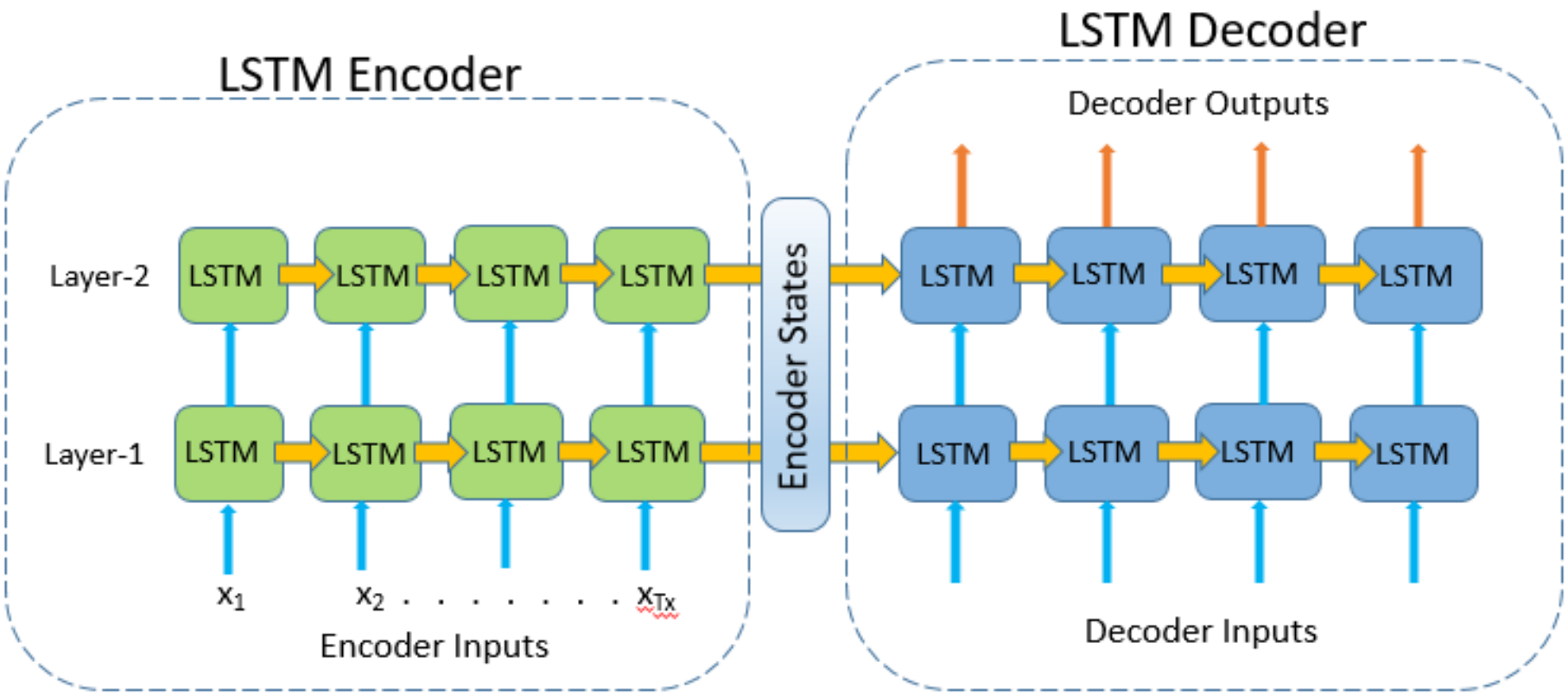


- Bidirectional recurrent neural networks(RNN) combine two independent RNNs
- Input sequence is fed in normal time order
- The outputs of the two networks are usually concatenated at each time step

Stacked RNN



Encoder-Decoder RNN



RNN Summary

- RNN uses memory to remember patterns in the sequence
  - Uses non-linear hidden states
- LSTM maintains separate cell states
  - Use gates to control the flow of information
    - Forget gate gets rid of irrelevant information
    - Selectively update cell state
    - Output gate returns a filtered version of the cell state
  - Backpropagation from C_t to C_{t-1} does not require matrix multiplication
    - Uninterrupted gradient flow
  - It is possible to use bidirectional/stacked LSTM (similar to vanilla RNN)
  - However, very hard to train