Recurrent Neural and Long-Short Memory Networks

Alejandro Veloz

How do we do sequential learning?

Some **special neural networks** designed to work on sequential data:

- RNN (Recurrent Neural Network) - the fundamental architecture.
- LSTM (Long Short-Term Memory) - a more advanced and complex RNN that overcomes some limitations of traditional RNNs.
- GRU (Gated Recurrent Unit) - a simpler but effective and faster version of LSTM.
- Transformers - the most powerful.

Recurrent Neural Network

State machines

A **state machine** is a description of a process (computational, physical, economic) in terms of its potential sequences of **states**.

The **state** of a system is defined to be **all you would need to know** about the system to predict its future trajectories.

Formally, we define a state machine as: $(\mathcal{S}, \mathcal{X}, \mathcal{Y}, s_0, f_s, f_o)$

$\mathcal{S}$, $\mathcal{X}$, $\mathcal{Y}$ are sets of possible states, inputs, and outputs, respectively.

$s_0 \in \mathcal{S}$ is the initial state of the machine.

$f_s : \mathcal{S} \times \mathcal{X} \to \mathcal{S}$ is a *transition function* (which takes an input and a previous state and produces a next state).

$f_o : \mathcal{S} \to \mathcal{Y}$ is an *output function* (which takes a state and produces an output).

State machines

The basic operation of the state machine is to start with state s_0, then iteratively compute for $t \geq 1$:

$$s_t = f_s(s_{t-1}, x_t)$$
$$y_t = f_o(s_t)$$

Given a sequence of inputs $x_1, x_2, \ldots$ the machine generates a sequence of outputs:

$$\underbrace{f_o(\underbrace{f_s(s_0, x_1)}_{s_1})}_{y_1}$$

State machines

The basic operation of the state machine is to start with state s_0, then iteratively compute for $t \geq 1$:

$$s_t = f_s(s_{t-1}, x_t)$$
$$y_t = f_o(s_t)$$

Given a sequence of inputs $x_1, x_2, \ldots$ the machine generates a sequence of outputs:

$$\underbrace{f_o(\underbrace{f_s(s_0, x_1)}_{s_1})}_{y_1}, \; \underbrace{f_o(\underbrace{f_s(f_s(s_0, x_1), x_2)}_{s_2})}_{y_2}, \ldots$$

Recurrent neural network (RNN)

A RNN is a state machine in which neural networks constitute the functions f_s and f_o:

$$s_t = f_s\Big(W^{sx}x_t + W^{ss}s_{t-1} + W_0^{ss}\Big)$$

$$y_t = f_o\Big(W^{o}s_t + W_0^{o}\Big).$$

- f_s and f_o are activation functions.
- The inputs, states, and outputs are all vector-valued:

  $x_t : \ell \times 1, \quad s_t : m \times 1, \quad y_t : v \times 1.$

- The weights in the network are:

  $W^{sx} : m \times \ell,\ W^{ss} : m \times m,$

  $W_0^{ss} : m \times 1, W^{o} : v \times m, \quad W_0^{o} : v \times 1,$

Remember that we apply f_s and f_o elementwise, unless f_o is a softmax activation.

Sequence-to-sequence learning

Sequence-to-sequence mapping can be viewed as a regression problem: *given an input sequence, the goal is to learn how to generate the corresponding output sequence.*

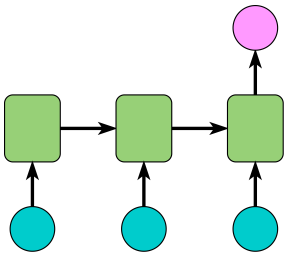
A training set has the form:

$$\left[\left(x^{(1)}, y^{(1)}\right), \ldots, \left(x^{(q)}, y^{(q)}\right)\right],$$

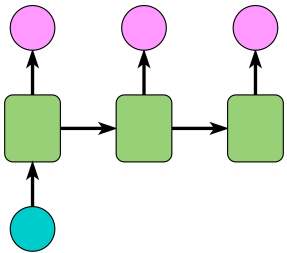
where:

- $x^{(i)}$ and $y^{(i)}$ are sequences of length $n^{(i)}$,
- the sequences within the *same pair* have the same length, and
- sequences in different pairs may have different lengths.

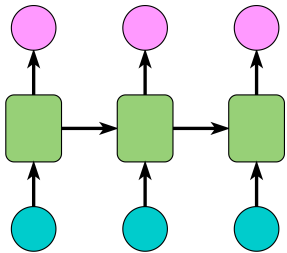
Types of sequence-to-sequence learning



Many to one

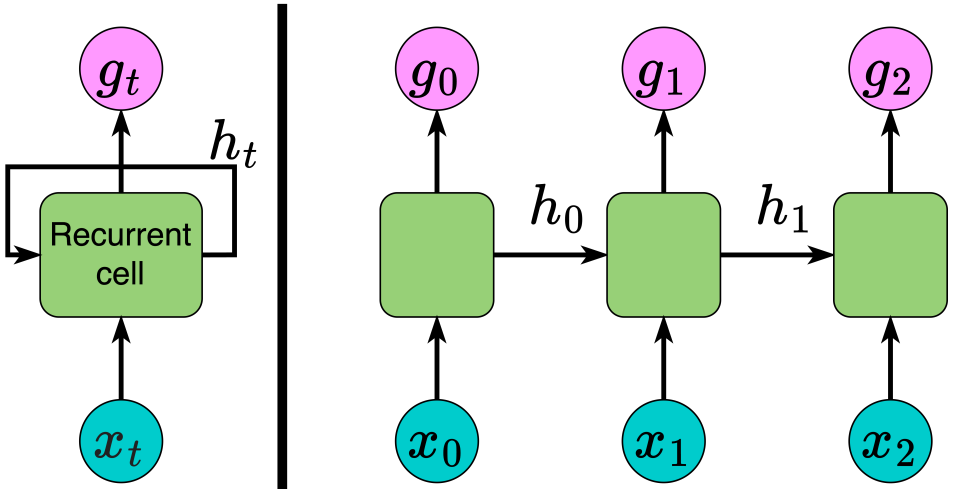


One to many

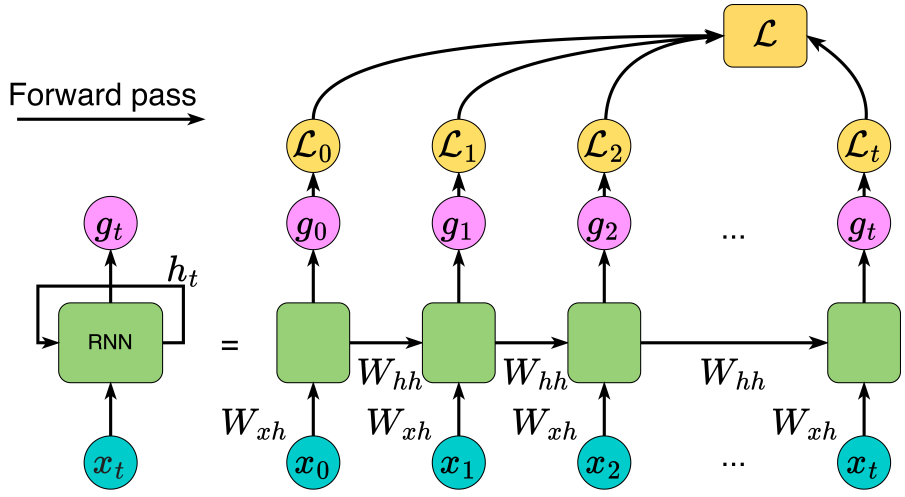


Many to many

RNN forward pass



RNN forward pass



Loss functions for sequence-to-sequence models

Many possible ways to define a **loss function** for sequences.

A common approach is to apply a per-element loss and sum over the sequence length. For example:

Let $g = [g_1, g_2, ...]$ be the predicted sequence produced by the RNN model, and $y = [y_1, y_2, ...]$ the ground truth sequence.

The sequence loss can then be defined as:

$$\mathcal{L}_{\text{seq}}\left(g^{(i)}, y^{(i)}\right) = \sum_{t=1}^{n^{(i)}} \mathcal{L}_{\text{elt}}\left(g_t^{(i)}, y_t^{(i)}\right).$$

Loss functions for sequence-to-sequence models

The **per-element loss** $\mathcal{L}_{\text{elt}}$ depends on the nature of the outputs y_t, e.g.

- Categorical cross-entropy for class labels.
- Squared loss for continuous targets.

Sequence-to-sequence model training

The model parameters are $W = (W^{sx}, W^{ss}, W^{o}, W_0^{ss}, W_0^{o})$.

The **training objective** is to minimize the average sequence loss across all training examples:

$$J(W) = \frac{1}{q} \sum_{i=1}^{q} \mathcal{L}_{\text{seq}}(\text{RNN}(x^{(i)}; W),\ y^{(i)}),$$

where $\text{RNN}(x; W)$ denotes the output sequence produced for input sequence x.

Types of loss functions

- **Element-wise losses:** sum a per-step loss such as **Negative Log-Likelihood (NLL)** for classification or **Mean Squared Error (MSE)** for regression.
- **Sequence-level losses:** add losses that capture properties of the *whole* sequence, such as:
  - **Connectionist Temporal Classification (CTC) loss** – for unaligned sequence data (e.g., speech recognition).
  - **Sequence-to-sequence (Seq2Seq) cross-entropy loss** – for discrete outputs, e.g. words.
  - **Edit-distance or BLEU-score inspired losses** – for structured predictions where sequence accuracy matters globally.
  - **Reinforcement-style sequence rewards** – for tasks like text generation, where quality is not purely per-element.

RNN as a language model

A *language model* is a sequence-to-sequence RNN trained on a token sequence $c = (c_1, c_2, \dots, c_k)$ and is used to predict the next token c_t (for $t \leq k$) given the previous $t - 1$ tokens:

$$c_t = \text{RNN}\Big((c_1, c_2, \dots, c_{t-1}); W\Big).$$

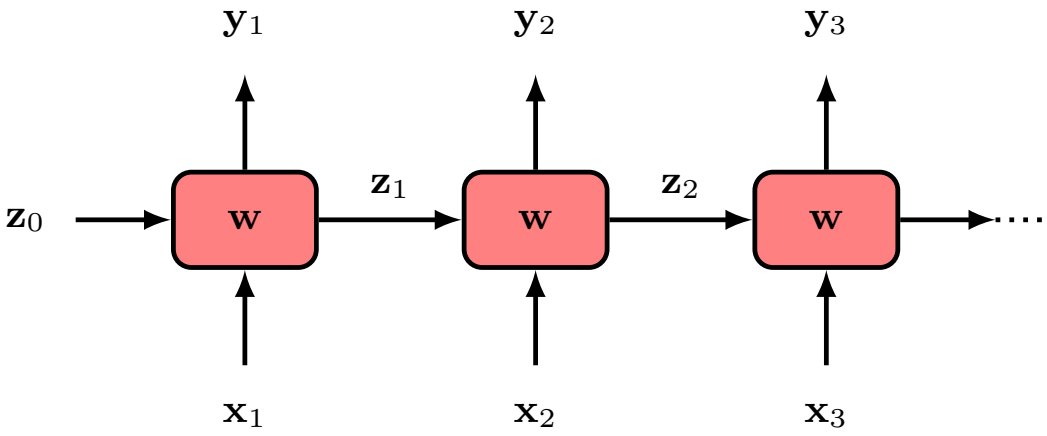
RNN as a language model

We can convert this to a sequence-to-sequence training problem by constructing a dataset of q different (x, y) sequence pairs.

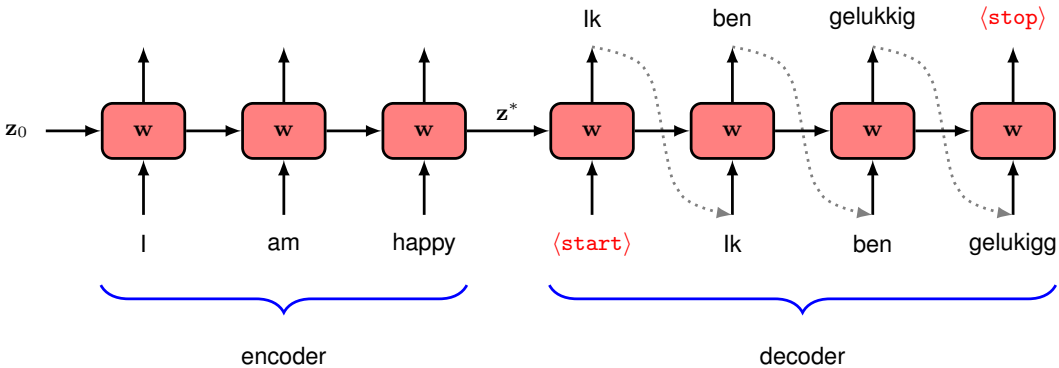
We introduce special tokens $\langle \text{start} \rangle$ and $\langle \text{end} \rangle$ to signal the beginning and end of the sequence:

$$x = (\langle \text{start} \rangle, c_1, c_2, \dots, c_k),$$
$$y = (c_1, c_2, \dots, \langle \text{end} \rangle).$$

RNN as a language model



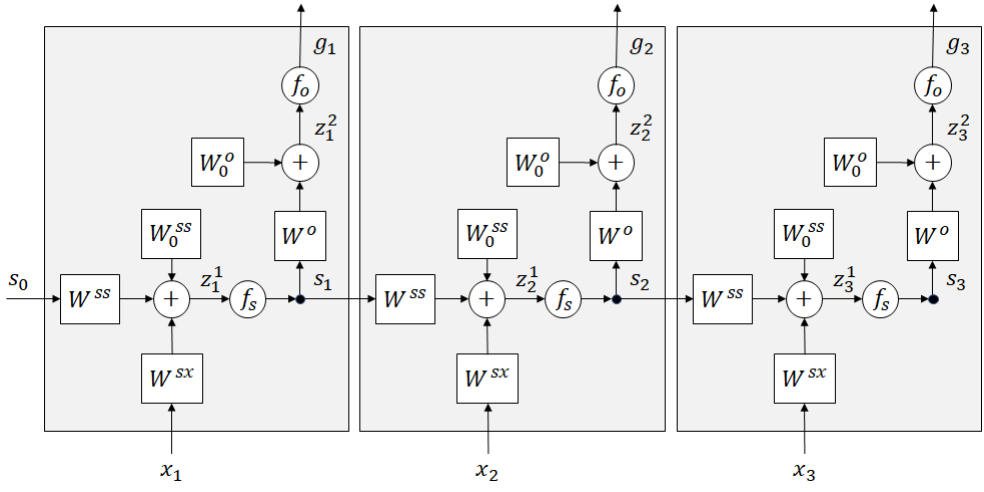
RNN as a language model



Back-propagation through time (BPTT)

Find a W to minimize J using gradient descent.

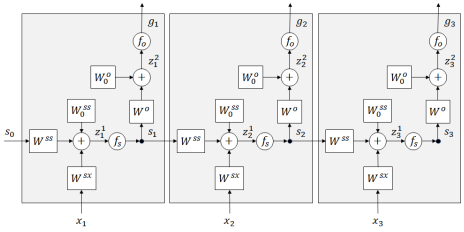
Unrolled RNN



Back-propagation through time (BPTT)

The **BPTT** process goes as follows:

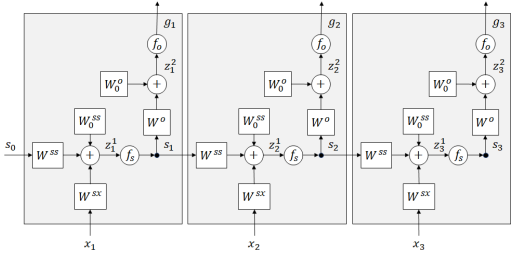
1. **Sampling:** Sample a training pair of sequences (x, y); let their length be n.
2. **Unrolling:** "Unroll" the RNN to length n (for example, $n = 3$), and initialize s_0:



The problem resembles performing back-propagation on a feed-forward network, except that the weight matrices are shared across time.
This is similar in spirit to CNN, where weights are reused spatially.

Back-propagation through time (BPTT)

3. **Forward Pass:** Compute the predicted output sequence g via:



$$z_t^1 = W^{sx}x_t + W^{ss}s_{t-1} + W_0^{ss}$$
$$s_t = f_s(z_t^1)$$
$$z_t^2 = W^o s_t + W_0^o$$
$$g_t = f_o(z_t^2)$$

Back-propagation through time (BPTT)

4. **Backward Pass:** Compute the gradients.

For both W^{ss} and W^{sx}, we need:

$$\frac{d\mathcal{L}_{\text{seq}}(g, y)}{dW} = \sum_{u=1}^{n} \frac{d\mathcal{L}_{\text{elt}}(g_u, y_u)}{dW}.$$

Let $\mathcal{L}_u = \mathcal{L}_{\text{elt}}(g_u, y_u)$. Using the total derivative (summing over all the ways W affects $\mathcal{L}_u$), we have:

$$\frac{d\mathcal{L}_{\text{seq}}}{dW} = \sum_{t=1}^{n} \frac{\partial s_t}{\partial W} \left(\frac{\partial \mathcal{L}_t}{\partial s_t} + \underbrace{\sum_{u=t+1}^{n} \frac{\partial \mathcal{L}_u}{\partial s_t}}_{\delta^{s_t}} \right).$$

δ^{s_t} represents the impact of state s_t on all future losses.

Define the *future loss* after step t as:

$$F_t = \sum_{u=t+1}^{n} \mathcal{L}_{\text{elt}}(g_u, y_u) \qquad \text{so that} \qquad \delta^{s_t} = \frac{\partial F_t}{\partial s_t}.$$

Note that $F_n = 0$ (hence $\delta^{s_n} = 0$).

Working backwards, for each t we have:

$$\delta^{s_{t-1}} = \frac{\partial s_t}{\partial s_{t-1}} \left[\frac{\partial \mathcal{L}_{\text{elt}}(g_t, y_t)}{\partial s_t} + \delta^{s_t} \right].$$

Using the chain rule, we write:

$$\frac{\partial \mathcal{L}_{\text{elt}}(g_t, y_t)}{\partial s_t} = \frac{\partial z_t^2}{\partial s_t} \frac{\partial \mathcal{L}_{\text{elt}}(g_t, y_t)}{\partial z_t^2},$$

and

$$\frac{\partial s_t}{\partial s_{t-1}} = \frac{\partial z_t^1}{\partial s_{t-1}} \frac{\partial s_t}{\partial z_t^1} = W^{ss\intercal} \frac{\partial s_t}{\partial z_t^1}.$$

Note that $\frac{\partial s_t}{\partial z_t^1}$ is formally an $m \times m$ diagonal matrix whose diagonal entries are $f_s^{'}(z_{t,i}^1)$ for $1 \leq i \leq m$.

We can represent this diagonal matrix as an $m \times 1$ vector $f_s^{'}(z_t^1)$. In that case, the product $W^{ss\mathsf{T}} * f_s^{'}(z_t^1)$ should be interpreted as multiplying each column of $W^{ss\mathsf{T}}$ by the corresponding entry of $f_s^{'}(z_t^1)$.

Putting everything together, we obtain:

$$\delta^{s_{t-1}} = W^{ss\mathsf{T}} \frac{\partial s_t}{\partial z_t^1} \left(W^{o\mathsf{T}} \frac{\partial \mathcal{L}_t}{\partial z_t^2} + \delta^{s_t} \right).$$

The gradients for the weight matrices are then given by

$$\frac{d\mathcal{L}_{\text{seq}}}{dW^{ss}} = \sum_{t=1}^{n} \frac{\partial z_t^1}{\partial W^{ss}} \frac{\partial s_t}{\partial z_t^1} \frac{\partial F_{t-1}}{\partial s_t},$$

$$\frac{d\mathcal{L}_{\text{seq}}}{dW^{sx}} = \sum_{t=1}^{n} \frac{\partial z_t^1}{\partial W^{sx}} \frac{\partial s_t}{\partial z_t^1} \frac{\partial F_{t-1}}{\partial s_t}.$$

The weight W^o is simpler because it does not affect future losses:

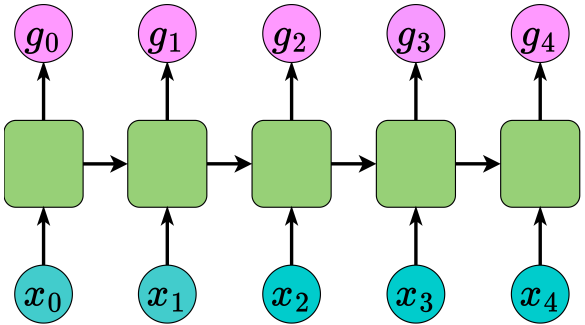
$$\frac{d\mathcal{L}_{\text{seq}}}{dW^o} = \sum_{t=1}^{n} \frac{\partial \mathcal{L}_t}{\partial z_t^2} \frac{\partial z_t^2}{\partial W^o}.$$

Assuming $\frac{\partial \mathcal{L}_t}{\partial z_t^2} = (g_t - y_t)$ (which holds for squared loss, softmax-NLL, etc.), then:

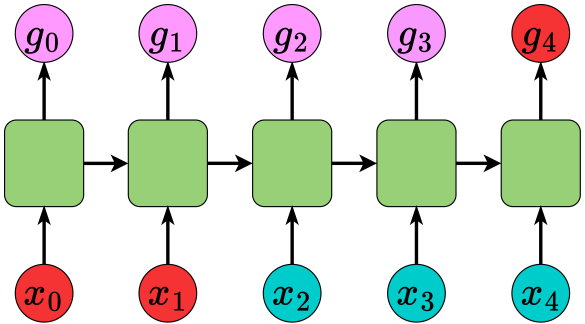
$$\frac{d\mathcal{L}_{\text{seq}}}{dW^o} = \sum_{t=1}^{n} (g_t - y_t)\, s_t^\intercal.$$

Derive the updates for the offsets W_0^{ss} and W_0^o.

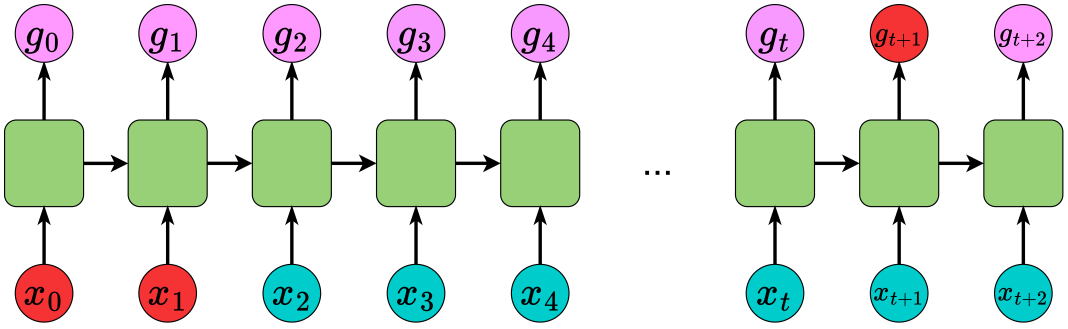
The problem of long-term dependences



The problem of long-term dependences

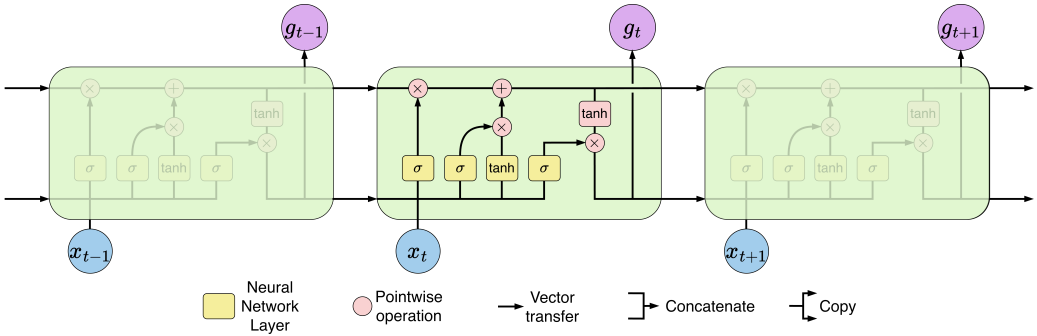


The problem of long-term dependences

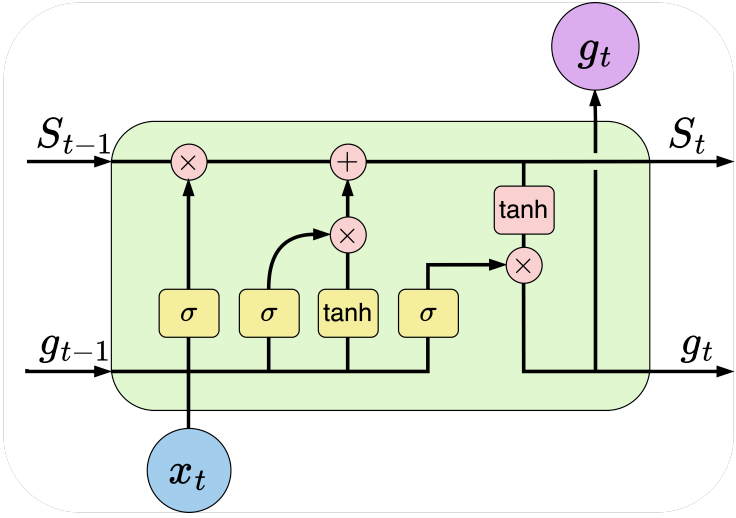


Long short-term memory

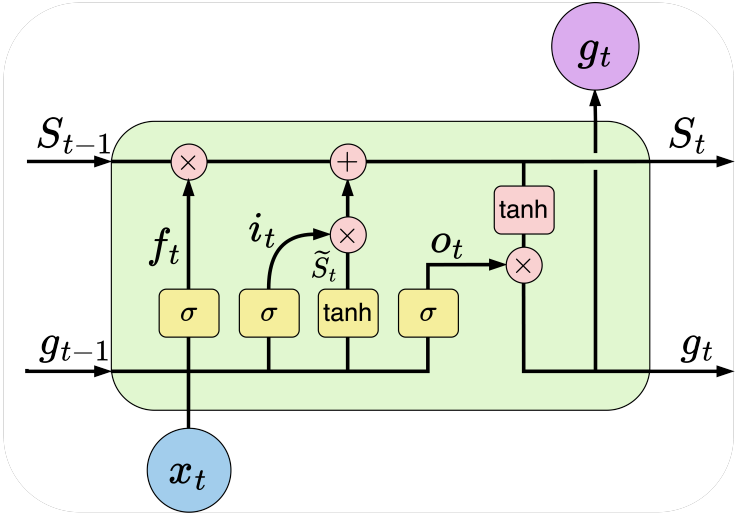
Long short-term memory



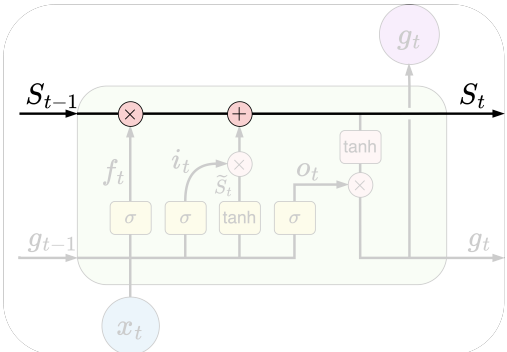
Long short-term memory



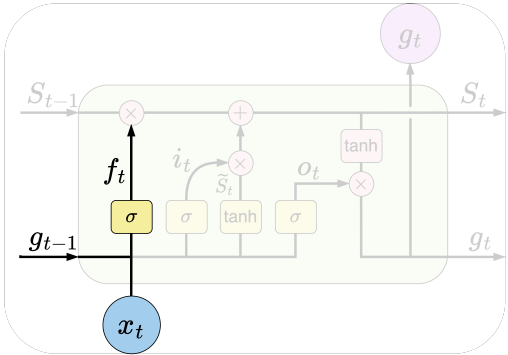
Long short-term memory



Long short-term memory



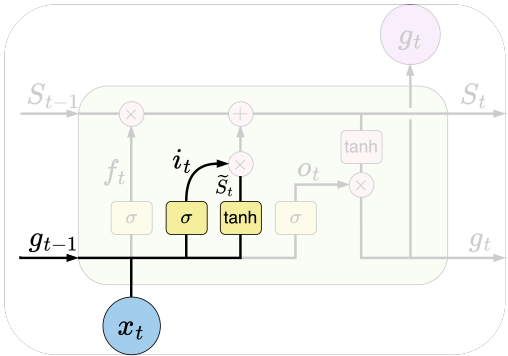
Long short-term memory



[forget gate layer]

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

Long short-term memory



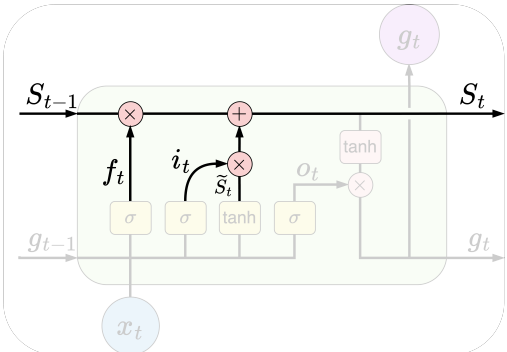
[input gate layer]

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

[new candidate states]

$$\tilde{S}_t = \tanh\left(W_S \cdot [g_{t-1}, x_t] + b_S\right)$$

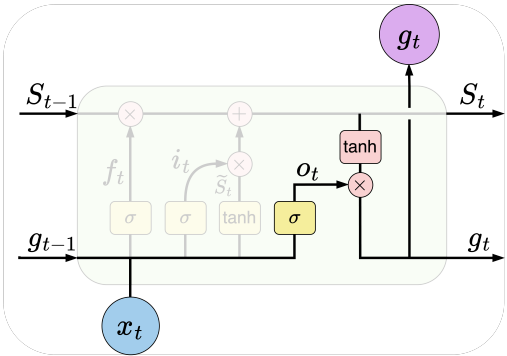
Long short-term memory



[State update]

$$S_t = f_t * S_{t-1} + i_t * \tilde{S}_t$$

Long short-term memory



[output]

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

$$g_t = o_t * \tanh\left(S_t\right)$$