TEXT ANALYTICS

Recurrent Neural Networks

Francesco Spinnato

* francesco.spinnato@unipi.it
University of Pisa



Where We Are

Word embeddings give every token a dense vector, and a text of T tokens becomes a **sequence of vectors**:

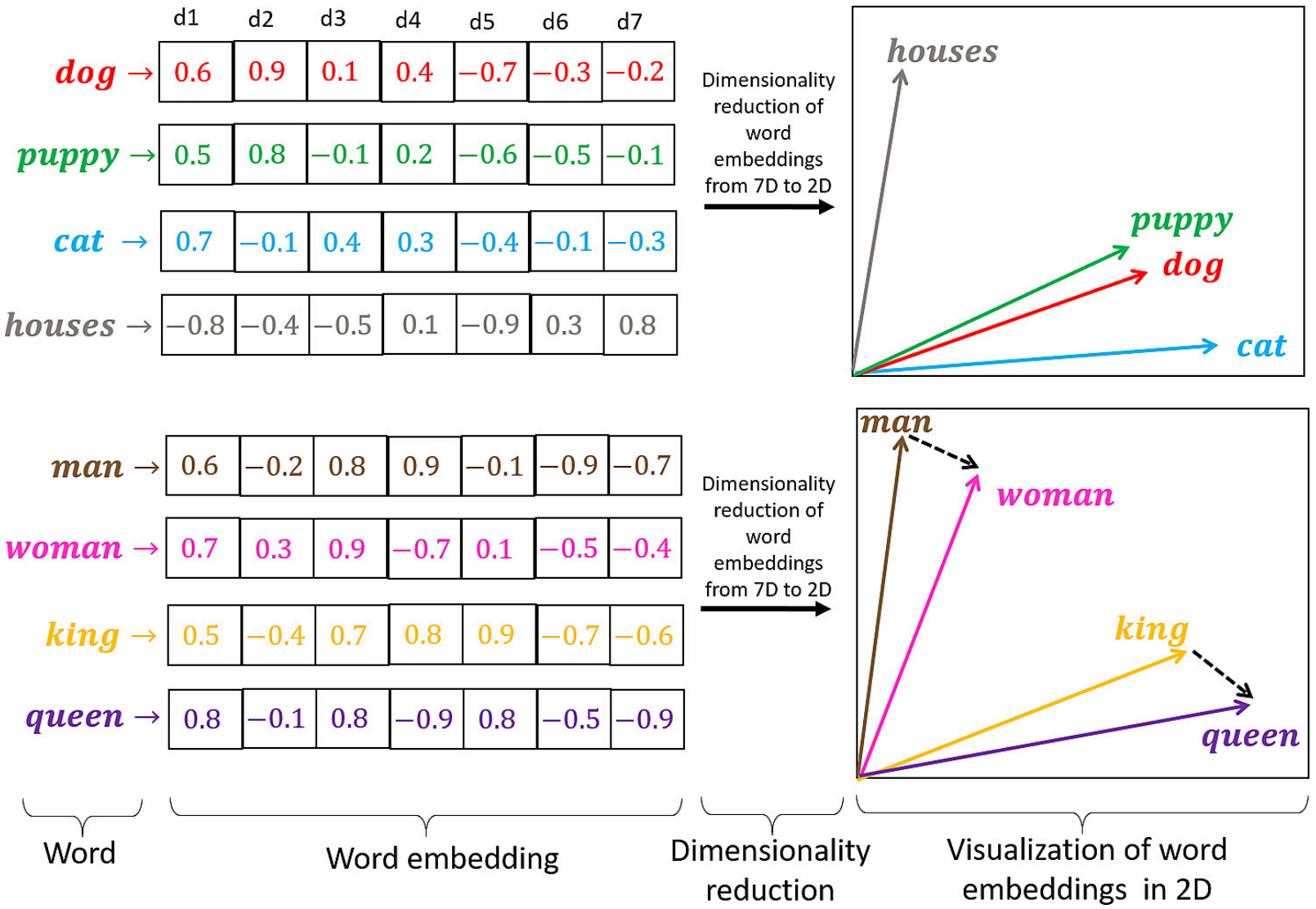
$$[w_1, \ldots, w_T] \longrightarrow [\mathbf{x}_1, \ldots, \mathbf{x}_T] \in \mathbb{R}^{T \times d}$$

where $\mathbf{x}_t = \mathbf{e}_{w_t} \in \mathbb{R}^d$.

- the embedding matrix is either **learned** together with the rest of the network, or **pretrained** (Word2Vec, GloVe) and then kept fixed or fine-tuned
- but these vectors are **static**
- and aggregating them trivially **forgets the order of the words** in a document

We tackle the last point: a model for text must **read the sequence**, in order, and keep track of what it has read.

Text Embeddings



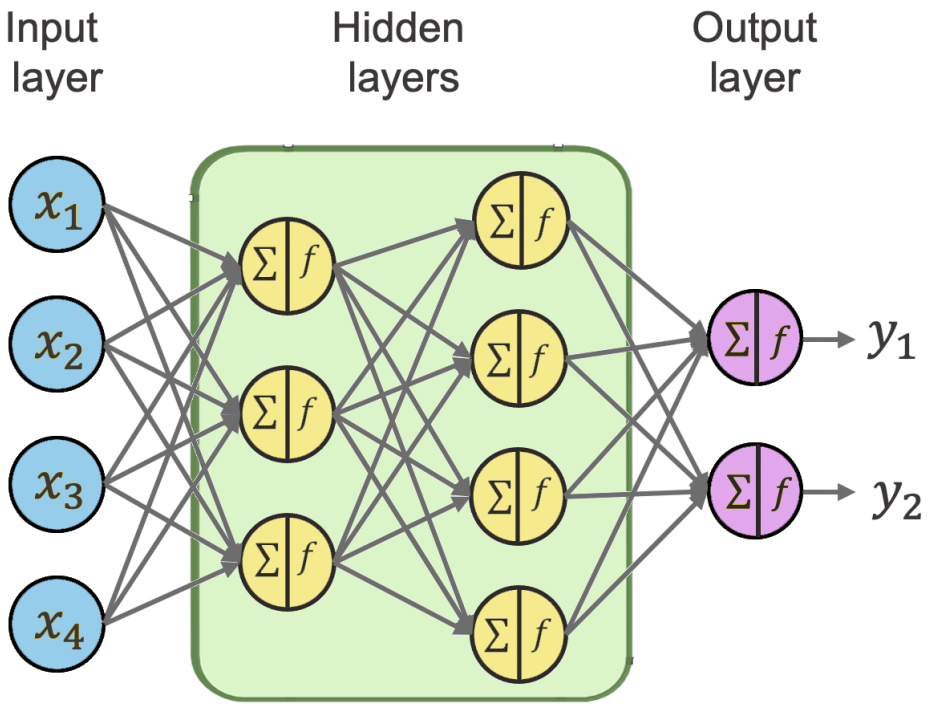
Why Not a Feedforward Network?

A feedforward network (or **vanilla** neural network), such as an MLP, is made of an **input layer**, **hidden layers**, an **output layer** and the **weights** between them.

It takes a **fixed-size** input, for example the vector of a text, and gives a **fixed-size** output:

$$\mathbf{h} = f(\mathbf{W}_{xh}\mathbf{x})$$

$$\hat{\mathbf{y}} = f(\mathbf{W}_{hy}\mathbf{h})$$



Why Not a Feedforward Network? The Problems

A feedforward network has problems with a text:

- **no parameter sharing**: a weight that detects a word in position 2 is a different weight from the one for position 3, so what is learned in one place is not reused in another
- **variable length**: it cannot handle texts of **different sizes**. Padding to a maximum wastes computation, and truncating loses words
- **no order**: there is no order in the input neuron, because tabular features are not sorted

Why Not a Feedforward Network? An Example

To answer *"What did you eat yesterday?"* a model must find *pizza* wherever it is in the text:

Yesterday I ate pizza

I ate pizza yesterday

What did you eat yesterday?

Tasks on Text Sequences

There are many tasks in which a **text**, a sequence of tokens, is the input or the output. **Recurrent neural networks (RNNs)** are built for them:

- **text classification**: a whole text gets one label. *"Decent effort. The plot could have been better."* $\rightarrow$ 3 stars
- **sequence labeling**: every token gets a label. *"Ygritte says Jon Snow knows nothing"* $\rightarrow$ the names of persons are *Ygritte* and *Jon Snow*
- **sequence to sequence**: a text is mapped to another text, possibly of a different length. *"chi non risica non rosica."* $\rightarrow$ *"If you don't take risks, you cannot win."*
- **generation**: the next token is predicted, again and again, to write a text. This is **language modeling**

Recurrent Neural Networks

Vanilla RNNs

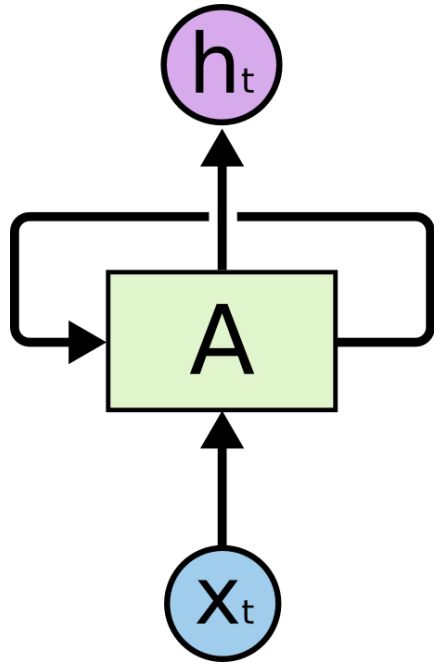
RNN

Recurrent neural networks (RNNs) are a class of neural networks in which **the state computed at the previous step is used as an input** at the current step.

They have a sort of “memory” that stores information about the past to inform predictions about the future.

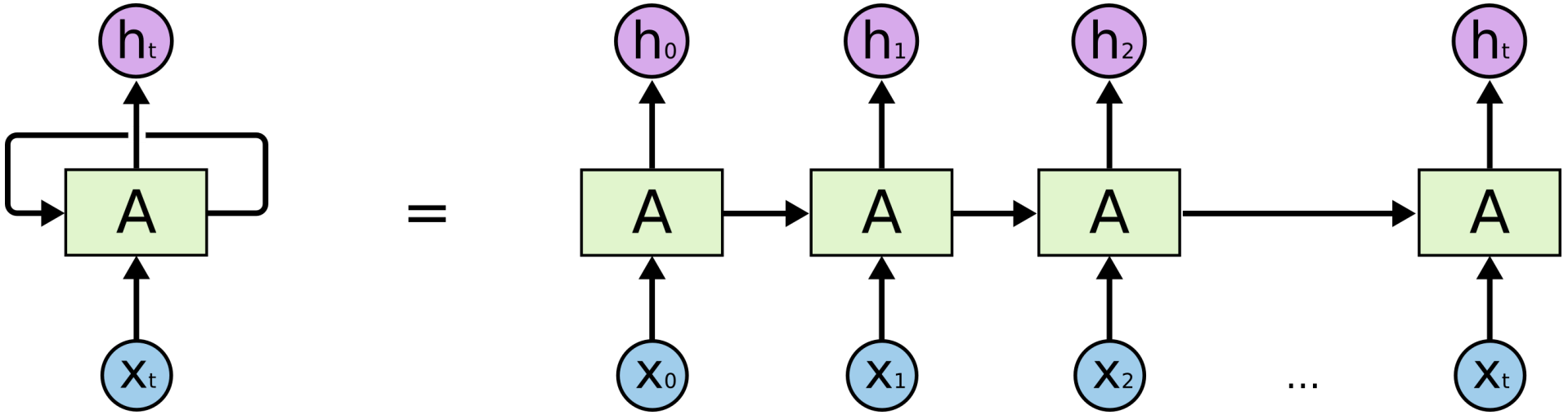
$$\mathbf{h}_t = f_{\mathbf{W}}(\mathbf{h}_{t-1}, \mathbf{x}_t)$$

- $\mathbf{x}_t$: input at step t, the embedding of the t-th token
- A: memory cell
- $\mathbf{h}_t$: new state
- $\mathbf{h}_{t-1}$: old state, with $\mathbf{h}_0 = \mathbf{0}$ before the first token
- $f_{\mathbf{W}}$: a function with parameters $\mathbf{W}$



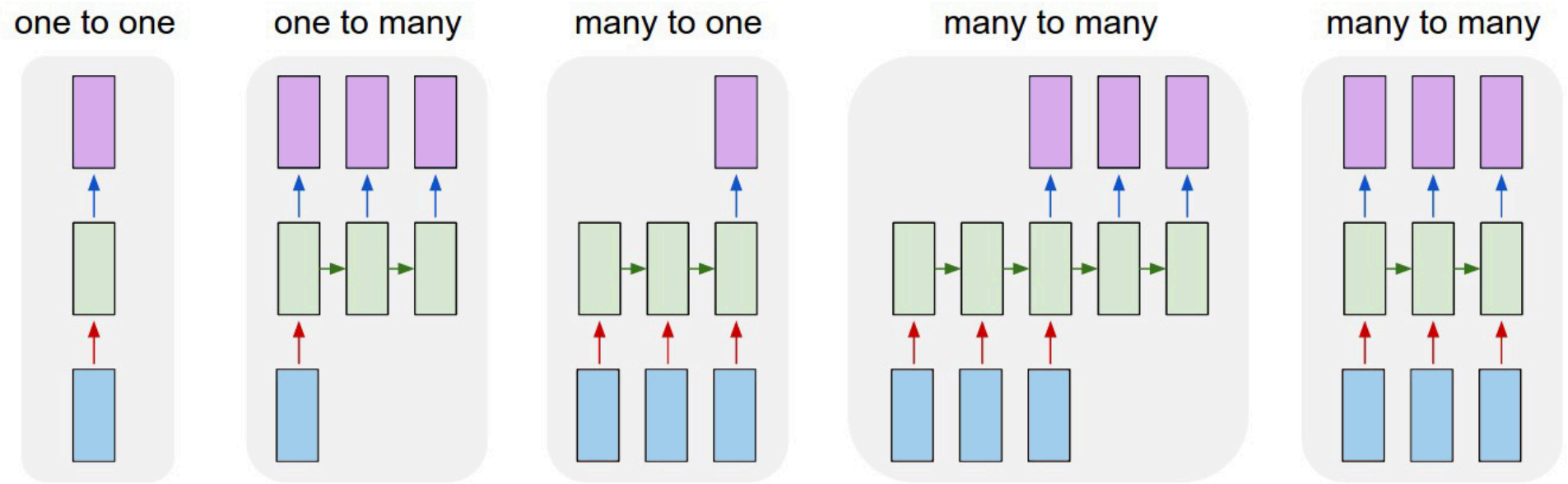
RNN - Unrolled

An RNN can be thought of as **multiple copies of the same network**, with the **same weights**, one for each token. Each copy passes its state to the next one.



Different Applications

There are different kinds of network structures depending on the task.



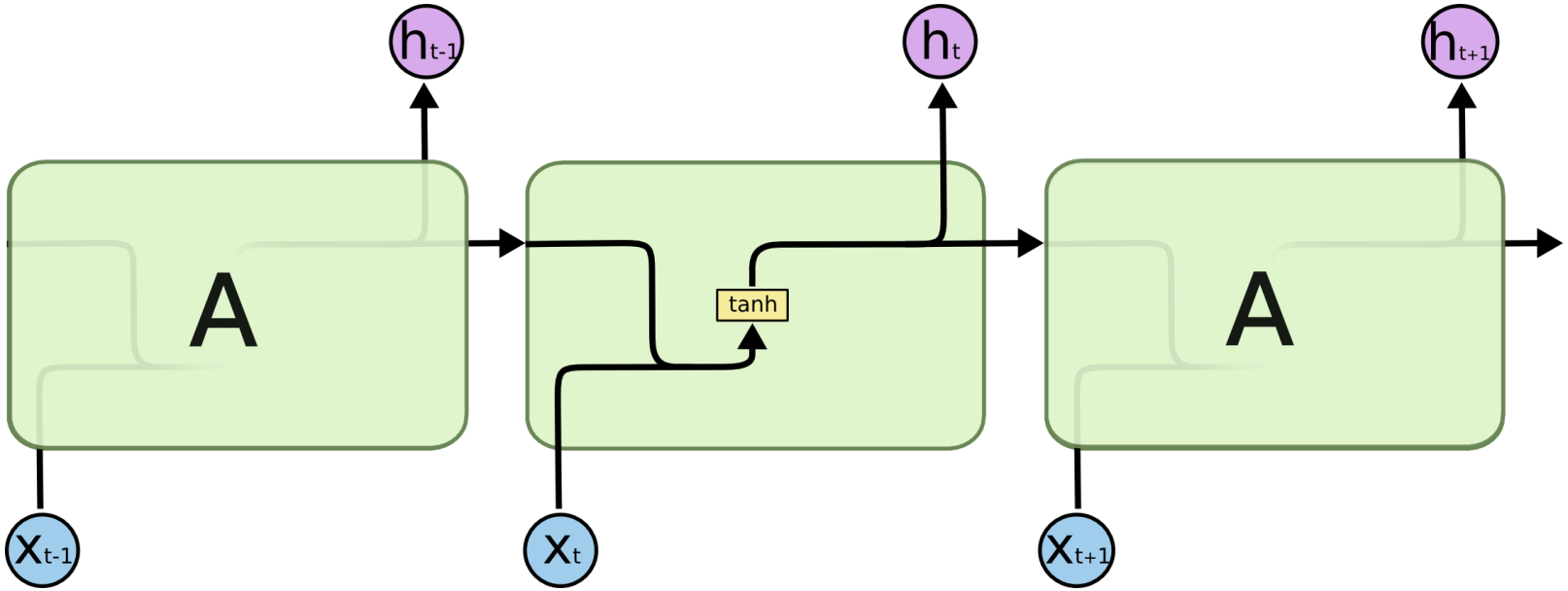
- **one to one**: vanilla neural network
- **one to many**: e.g. image captioning (image → sequence of words)
- **many to one**: e.g. sentiment classification (sequence of words → sentiment)
- **many to many**: e.g. machine translation (seq of words → seq of words)

Vanilla RNN

In the **vanilla RNN**, the function $f_{\mathbf{W}}$ is a single layer with a $\tanh$ activation:

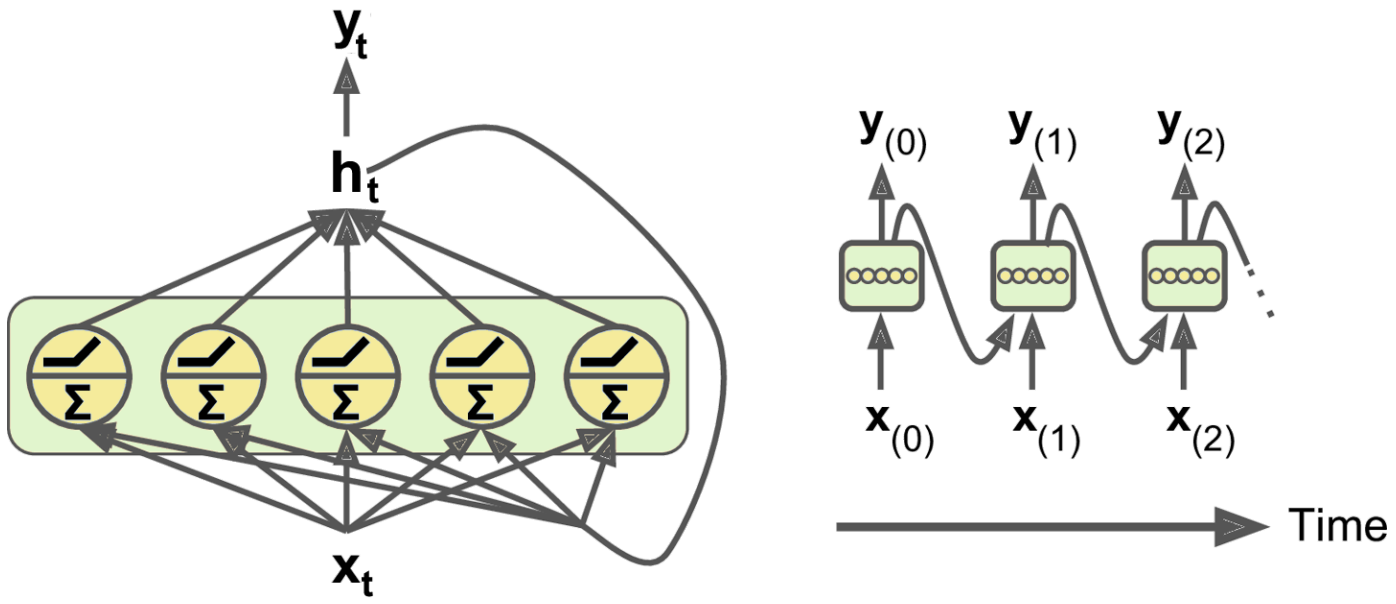
$$\boxed{\mathbf{h}_t = \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t)}$$

$\mathbf{W}_{xh}$ maps the input to the hidden state, $\mathbf{W}_{hh}$ maps the previous state to the new one. The same $\mathbf{W}_{xh}$ and $\mathbf{W}_{hh}$ are used at every step.



Output

The hidden state $\mathbf{h}_t$ is a vector: the memory cell has **multiple neurons.**

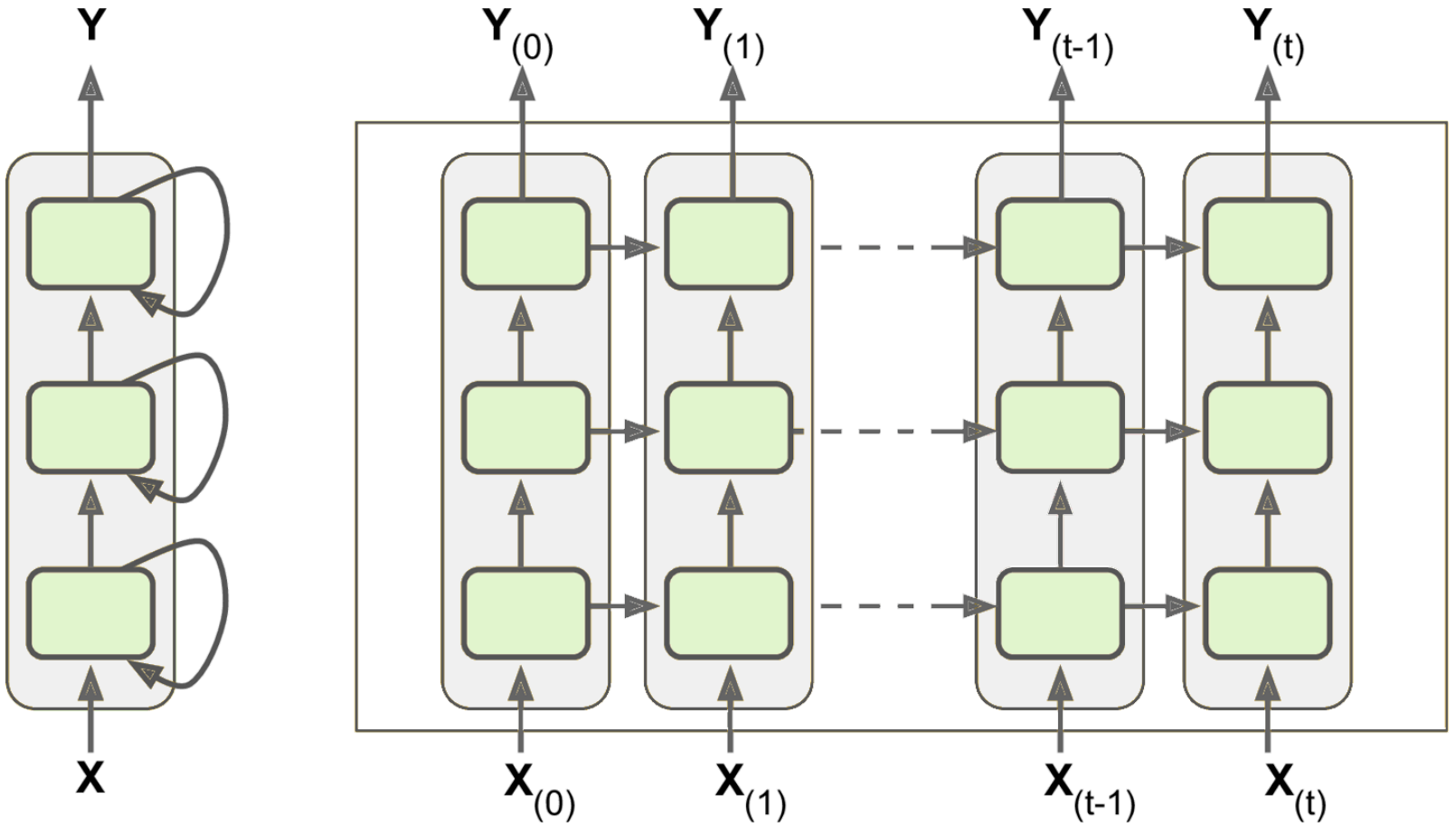


The output of an RNN is $\hat{\mathbf{y}}_t = \mathbf{W}_{hy}\mathbf{h}_t$. Depending on the task, an output can be read from the state:

- **at every step**
- or only **at the end**

Deep RNNs

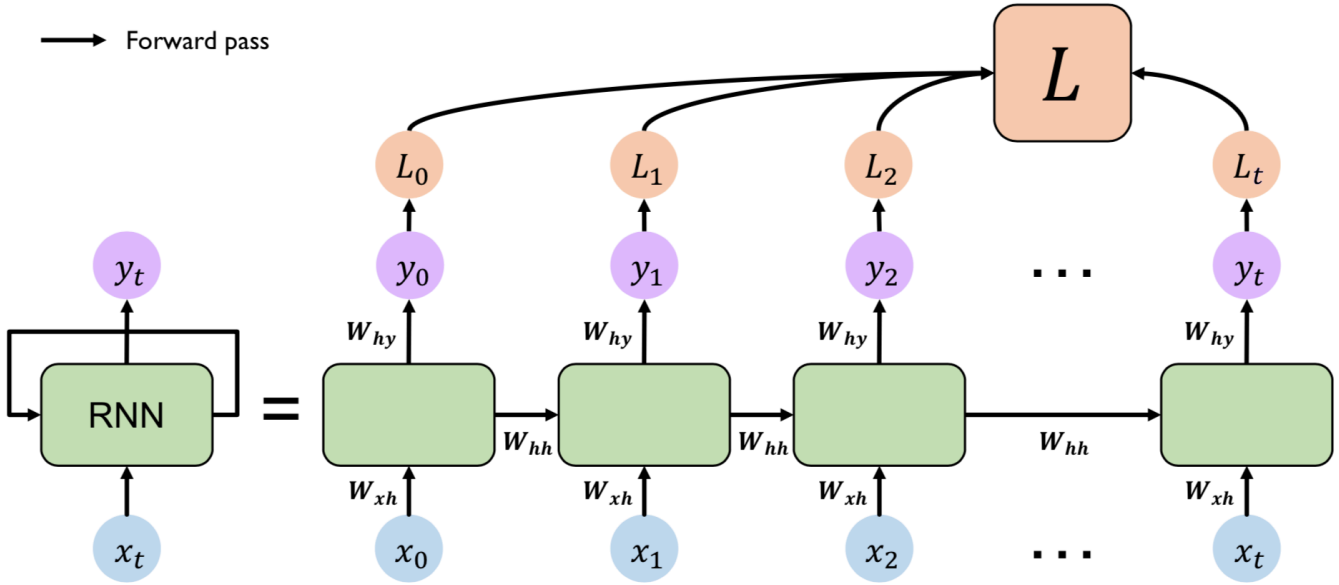
A **deep RNN** can be built by stacking multiple memory cells.



Training: Forward Pass

In the **forward pass**, each output $\hat{y}_t$ is compared to the ground truth y_t with a loss $\mathcal{L}$. The final loss is just the sum of all losses at each t.

$$\mathcal{L}(y, \hat{y}) = \sum_{t=0}^{T} \mathcal{L}_t(y_t, \hat{y}_t)$$



Example: RNN Forward

We consider a simple RNN with an **input of dimension 2**, and a **hidden state of dimension 1** (1 hidden unit).

$$\mathbf{W}_{hh} = [0.5], \quad \mathbf{W}_{xh} = [1 \quad -1]$$

The hidden state update is:

$$h_0 = 0, \quad h_t = \mathrm{ReLU}\left(0.5\, h_{t-1} + [1 \quad -1]\mathbf{x}_t\right)$$

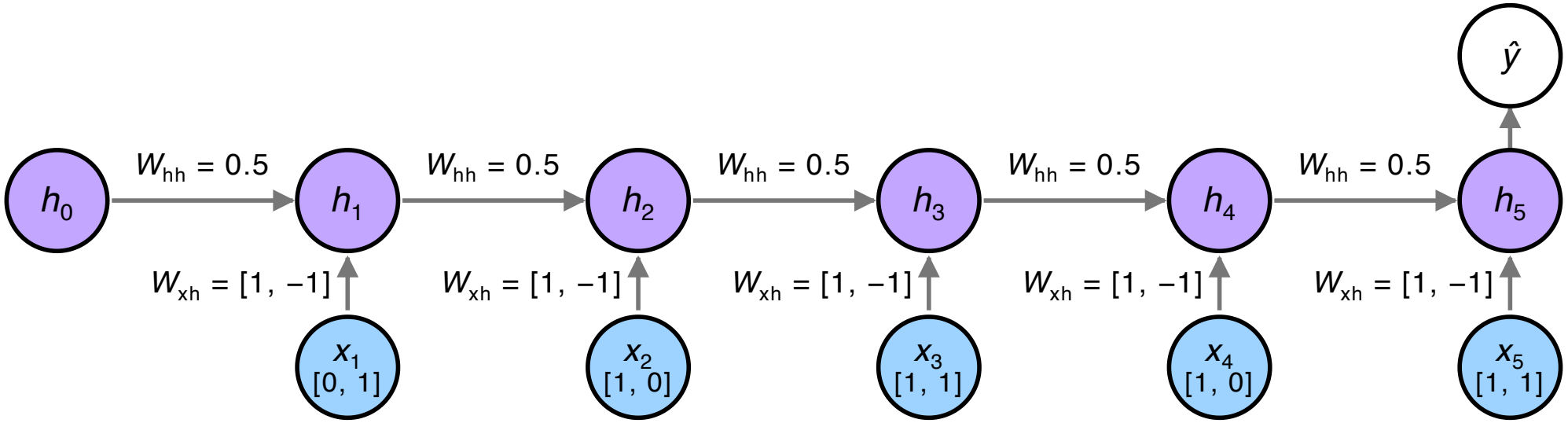
Our sequence is:

$$\mathbf{x}_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \mathbf{x}_3 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \mathbf{x}_4 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \mathbf{x}_5 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

In this toy example h is a scalar but in general it could be a vector. We use ReLU for simplicity.

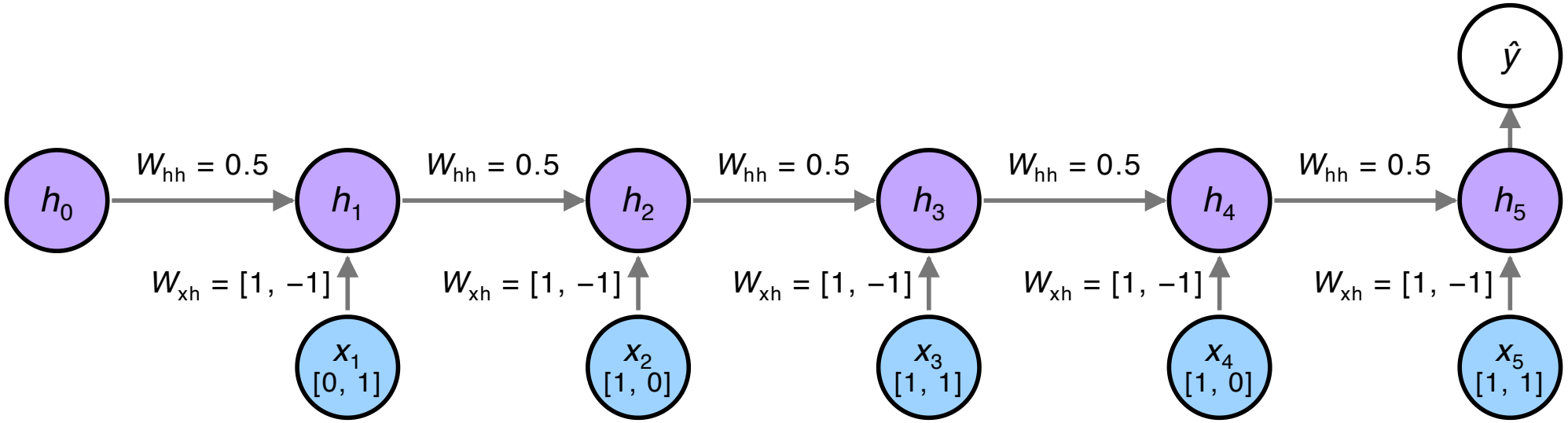
Example: RNN Forward $(t = 1)$

$$
\begin{aligned}
h_1 &= \mathrm{ReLU}\left(0.5 \cdot 0 + \begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix}\right) \\
&= \mathrm{ReLU}(0.5 \cdot 0 + 1 \cdot 0 + (-1) \cdot 1) \\
&= \mathrm{ReLU}(-1) = 0
\end{aligned}
$$



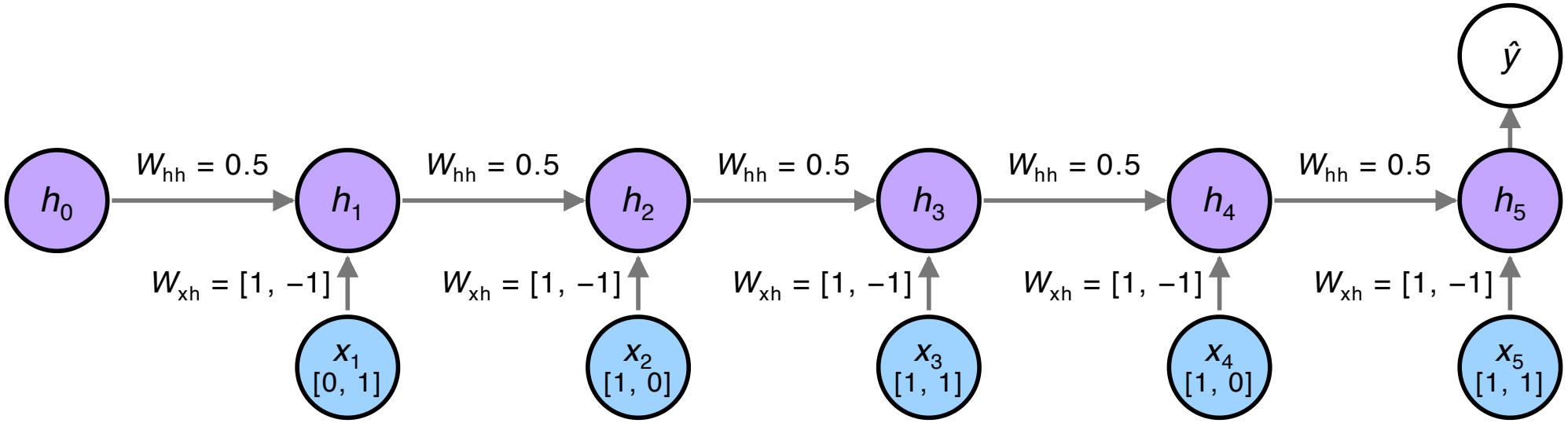
Example: RNN Forward $(t = 2)$

$$
\begin{aligned}
h_2 &= \mathrm{ReLU}\left(0.5 \cdot h_1 + \begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right) \\
&= \mathrm{ReLU}\left(0.5 \cdot 0 + 1 \cdot 1 + (-1) \cdot 0\right) \\
&= \mathrm{ReLU}(1) = 1
\end{aligned}
$$



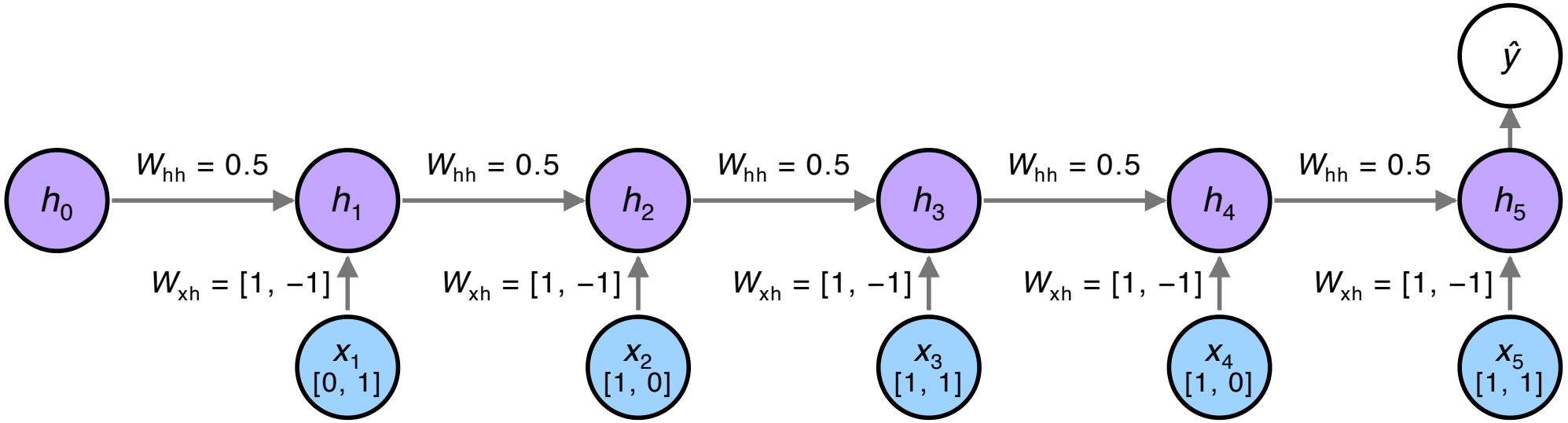
Example: RNN Forward $(t = 3)$

$$
\begin{aligned}
h_3 &= \mathrm{ReLU}\left(0.5 \cdot h_2 + \begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}\right) \\
&= \mathrm{ReLU}\left(0.5 \cdot 1 + 1 \cdot 1 + (-1) \cdot 1\right) \\
&= \mathrm{ReLU}(0.5) = 0.5
\end{aligned}
$$



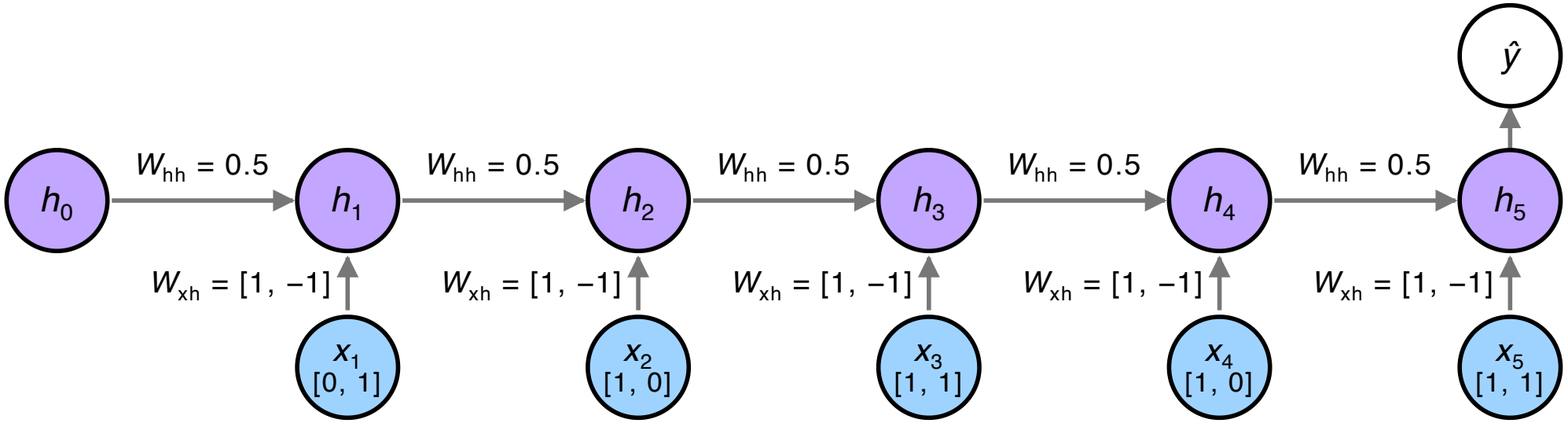
Example: RNN Forward $(t = 4)$

$$
\begin{aligned}
h_4 &= \mathrm{ReLU}\left(0.5 \cdot h_3 + \begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right) \\
&= \mathrm{ReLU}\left(0.5 \cdot 0.5 + 1 \cdot 1 + (-1) \cdot 0\right) \\
&= \mathrm{ReLU}(1.25) = 1.25
\end{aligned}
$$



Example: RNN Forward $(t = 5)$

$$
\begin{aligned}
h_5 &= \mathrm{ReLU}\left(0.5 \cdot h_4 + \begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}\right) \\
&= \mathrm{ReLU}\left(0.5 \cdot 1.25 + 1 \cdot 1 + (-1) \cdot 1\right) \\
&= \mathrm{ReLU}(0.625) = 0.625
\end{aligned}
$$



Example: RNN Forward Output

If our RNN is a many-to-one network, the last hidden state is used to compute the label of the whole text, for example its sentiment, e.g. with a sign activation function like in the perceptron.

Assuming $\mathbf{W}_{hy} = [1]$ for simplicity:

$$\hat{y} = \text{sign}(1 \cdot 0.625) = 1$$

So the whole text is assigned to the class $+1$. We just applied the forward pass in a **recurrent perceptron**.

Example: RNN Multiclass

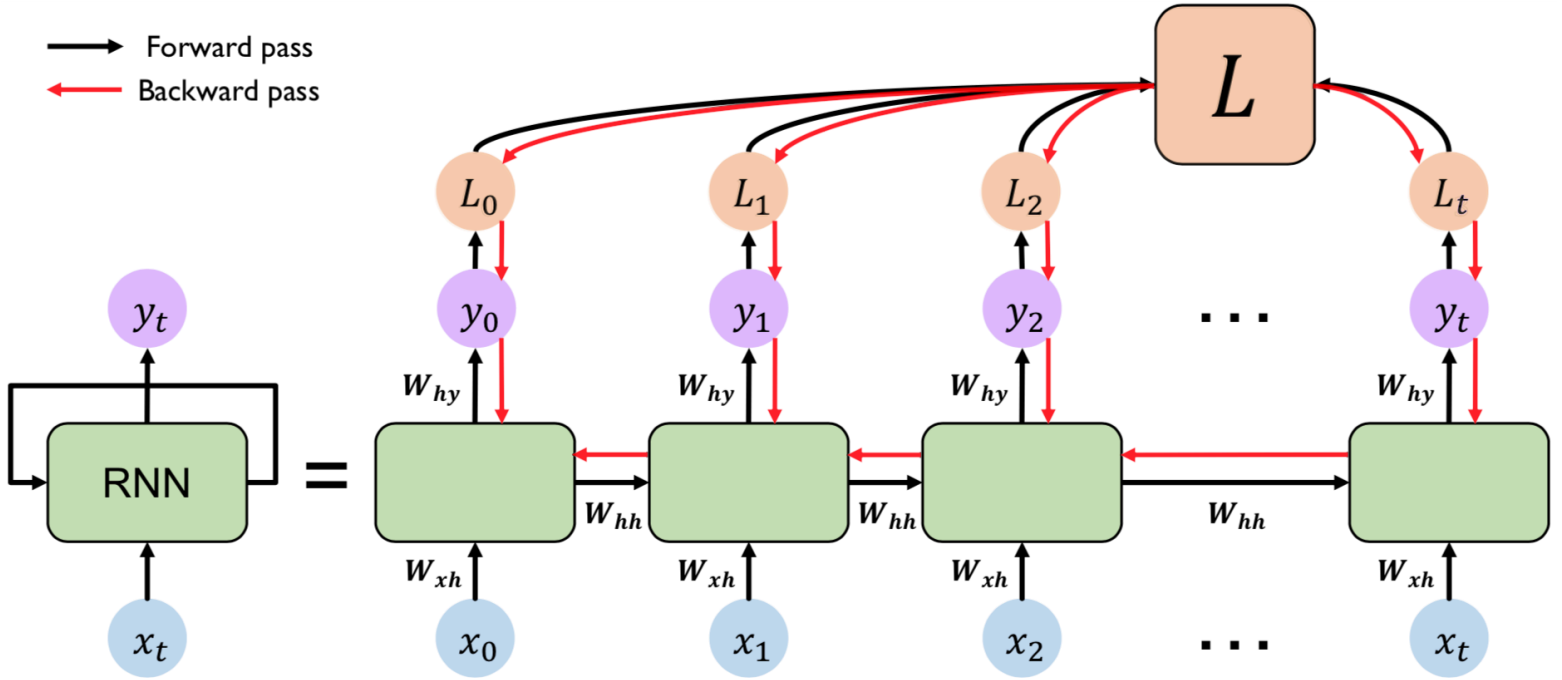
The same RNN can classify a text into K **classes**, for example $K = 3$ sentiments: positive, neutral and negative. The last hidden state, $h_5 = 0.625$, goes through a **softmax layer**: one score for each class, and then probabilities that sum to 1.

$$\mathbf{z} = \mathbf{W}_{hy}\, h_5 = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} 0.625 = \begin{bmatrix} 0.625 \\ 0 \\ -0.625 \end{bmatrix} \qquad \hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{z}) = \begin{bmatrix} 0.549 \\ 0.294 \\ 0.157 \end{bmatrix}$$

With $d_h = 1$ the matrix $\mathbf{W}_{hy}$ is 3×1. The text is assigned to the class with the highest probability, the first one.

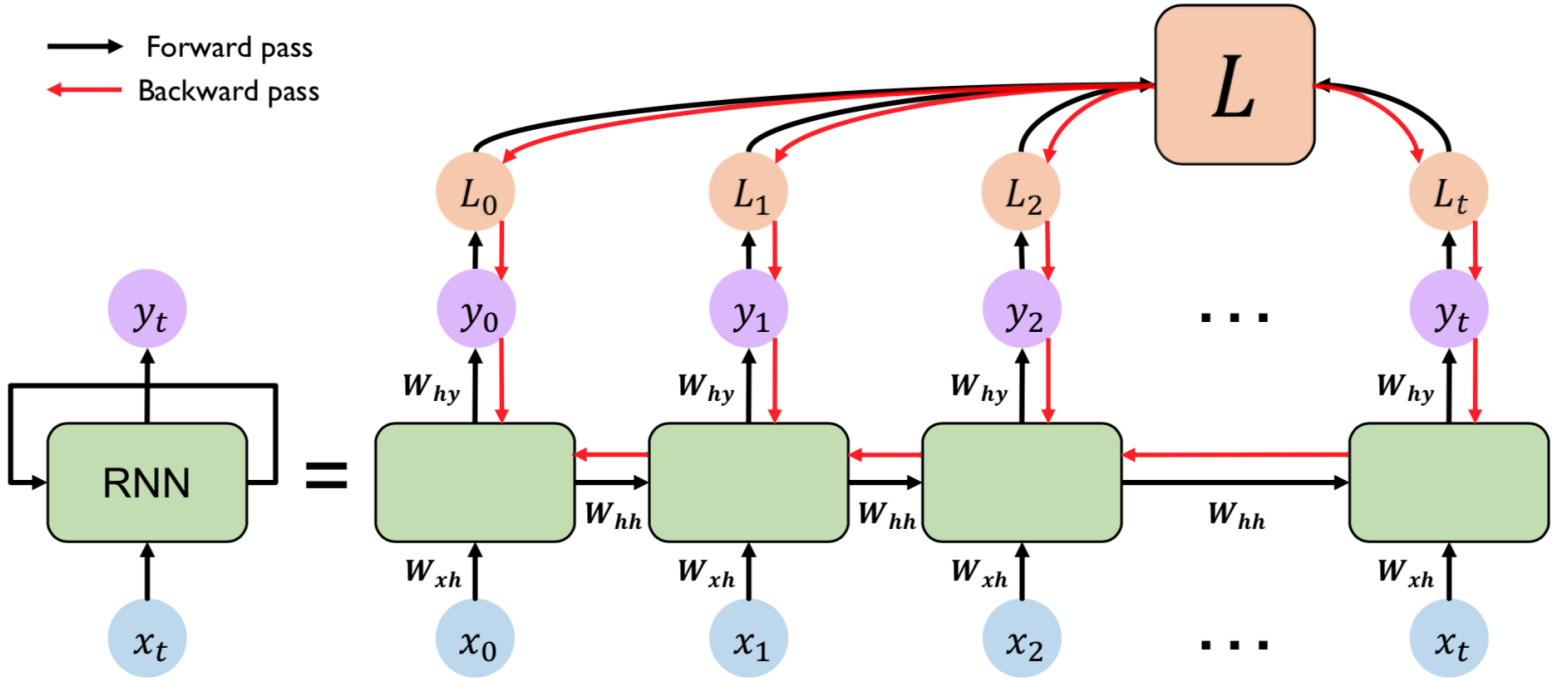
Training: Backward Pass

RNNs are trained using **backpropagation through time**. Similar to standard MLPs, we compute the gradient of the loss with respect to the weights. This tells us how changing a weight would change the total loss.



Training: Backward Pass

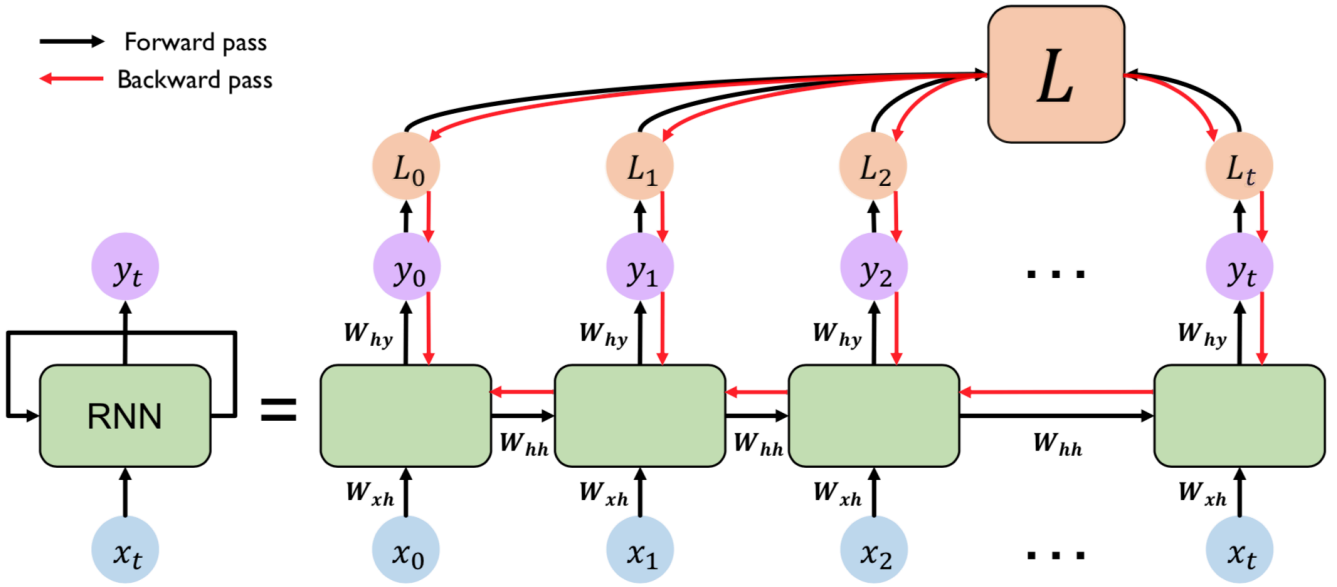
But the same weights are reused at every time step. For example, $\mathbf{W}_{hh}$ is used repeatedly to pass information from one hidden state to the next. So when we update the weights, we must account for their effect at every t.



Training: Backward Pass

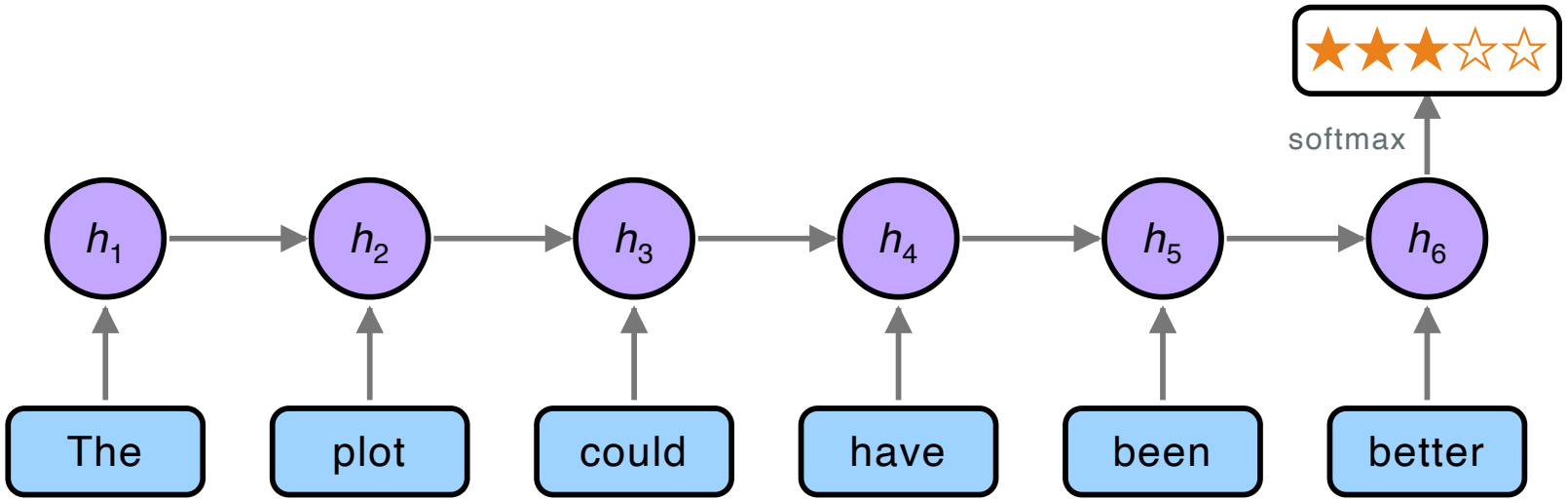
To know how much a weight contributed, we look at every time step t, then trace how that error flowed backward through all previous hidden states h_k.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \sum_{t=0}^{T} \sum_{k=0}^{t} \frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}}$$



Text Classification, Many to One

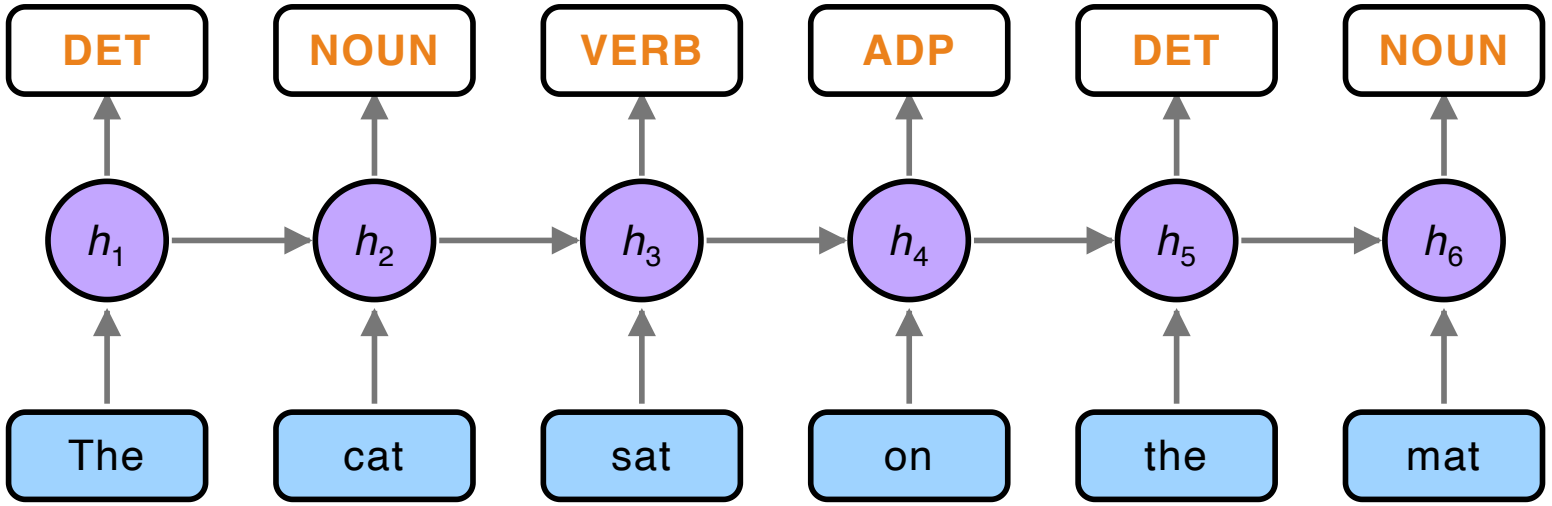
To classify a text, the **last hidden state**, the summary of the whole text, goes through a softmax layer that gives the class probabilities:



The network is trained with the **cross-entropy** loss.

Sequence Labeling, Many to Many

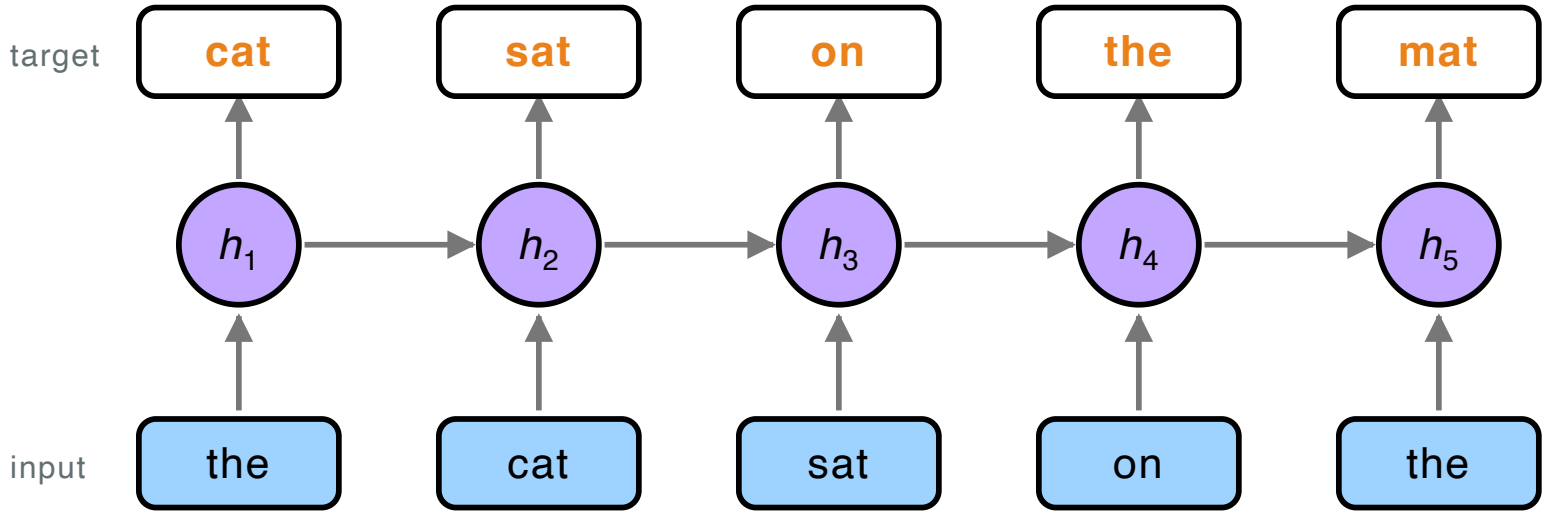
In **sequence labeling** every token gets a label, for example its **part of speech**: the grammatical category of the word (determiner, noun, verb, preposition). The RNN gives an output at **every step**:



The same idea finds the names of people and places (named entity recognition). Each tag depends only on the words **read so far**: the words that **follow** are not seen.

An RNN Language Model

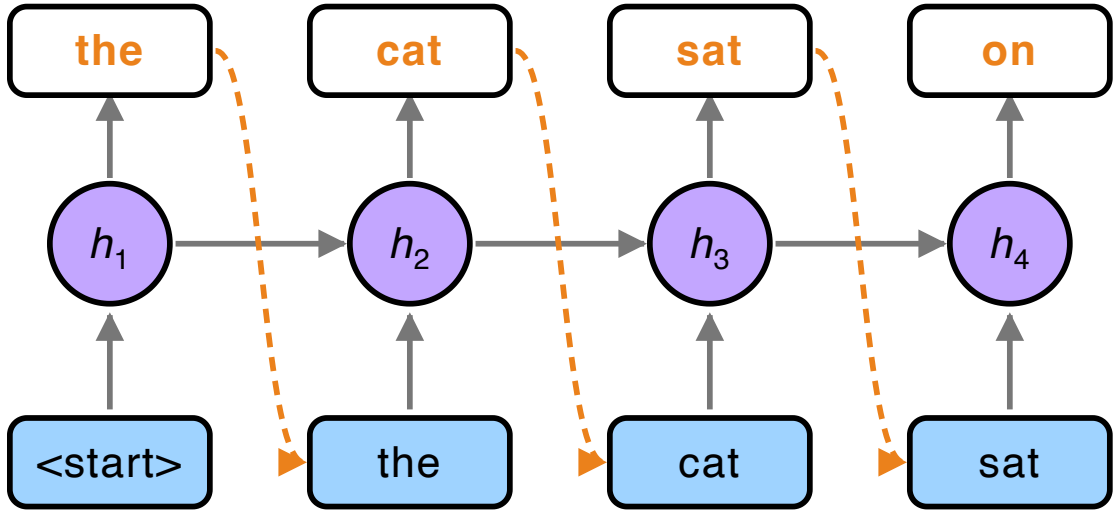
A **language model** predicts the next token given the previous ones. At every step the RNN gives a probability to **each word of the vocabulary**, and its state remembers all the text read so far.



In training, the input at each step is the **true** token of the text, and the **target** is the next one (**teacher forcing**). The error is counted at every step.

Generating Text

A language model writes a text **one token at a time**. At every step it **chooses** the next token from its probabilities and **feeds it back** as the next input:



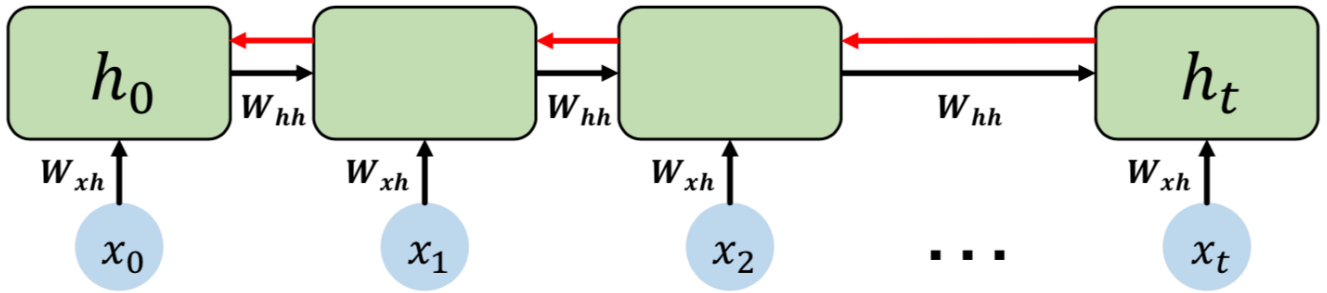
It starts from a start token and stops at an end token. In generation the inputs are the model's **own** choices, so a wrong choice changes everything that follows.

Gradients flow

In the backward pass, the error of step t reaches an earlier step k through a product with **one factor for every step in between:**

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} = \prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}$$

- If the factors are smaller than 1, the product goes to zero: **vanishing** gradients.
- If they are larger than 1, it grows without bound: **exploding** gradients.

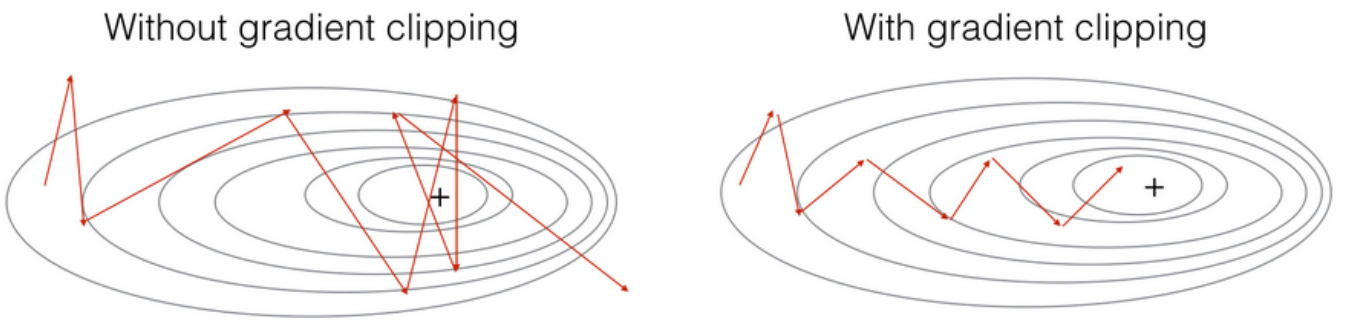


Exploding Gradients

When the gradient grows extremely large, it causes an **overflow** (NaN values). This is easy to detect, and is solved by **gradient clipping**:

$$\textbf{Given } \mathbf{g} = \frac{\partial \mathcal{L}}{\partial \mathbf{W}}, \textbf{ if } \|\mathbf{g}\| \geq \text{threshold}, \textbf{ then } \mathbf{g} = \mathbf{g} \frac{\text{threshold}}{\|\mathbf{g}\|}$$

In practice, whenever the norm of the gradient exceeds a threshold, the gradient is **scaled back to the threshold**, without changing its direction.



Vanishing Gradients

When the gradient goes to zero, the weights are hardly updated by what happened **far away** in the text: the network learns the **local** patterns, and misses long dependencies like the agreement between *cats* and *are* in *"The cats that the dog chased all day long are tired"*.

Possible solutions are:

- a smart **initialization** of $\mathbf{W}_{hh}$ (the identity matrix);
- **ReLU** instead of $\tanh$: its derivative is 0 or 1;
- **truncated backpropagation**: the gradient flows back only for a window of steps;
- **gating** (next section).

Past and Future Context

A standard RNN builds its state at step t using only the **past**, the words read so far. But the right interpretation of a word often depends on the words that **follow**. In:

"The bank of the river was muddy"

at *bank* the model has read only *The*.

This matters when the **whole text is available** (offline), as in sequence labeling and named entity recognition.

Bidirectional RNNs (BiRNN)

A **bidirectional RNN** runs two RNNs:

$$\overrightarrow{\mathbf{h}}_t = f(\overrightarrow{\mathbf{h}}_{t-1}, \mathbf{x}_t) \qquad \overleftarrow{\mathbf{h}}_t = f(\overleftarrow{\mathbf{h}}_{t+1}, \mathbf{x}_t)$$

Their outputs are combined (often by concatenation):

$$\mathbf{h}_t = [\overrightarrow{\mathbf{h}}_t ; \overleftarrow{\mathbf{h}}_t]$$

Limitation: requires the full sequence (not suitable for strict online/streaming inference).

Recurrent Neural Networks

Gated RNNs

Long Short-Term Memory

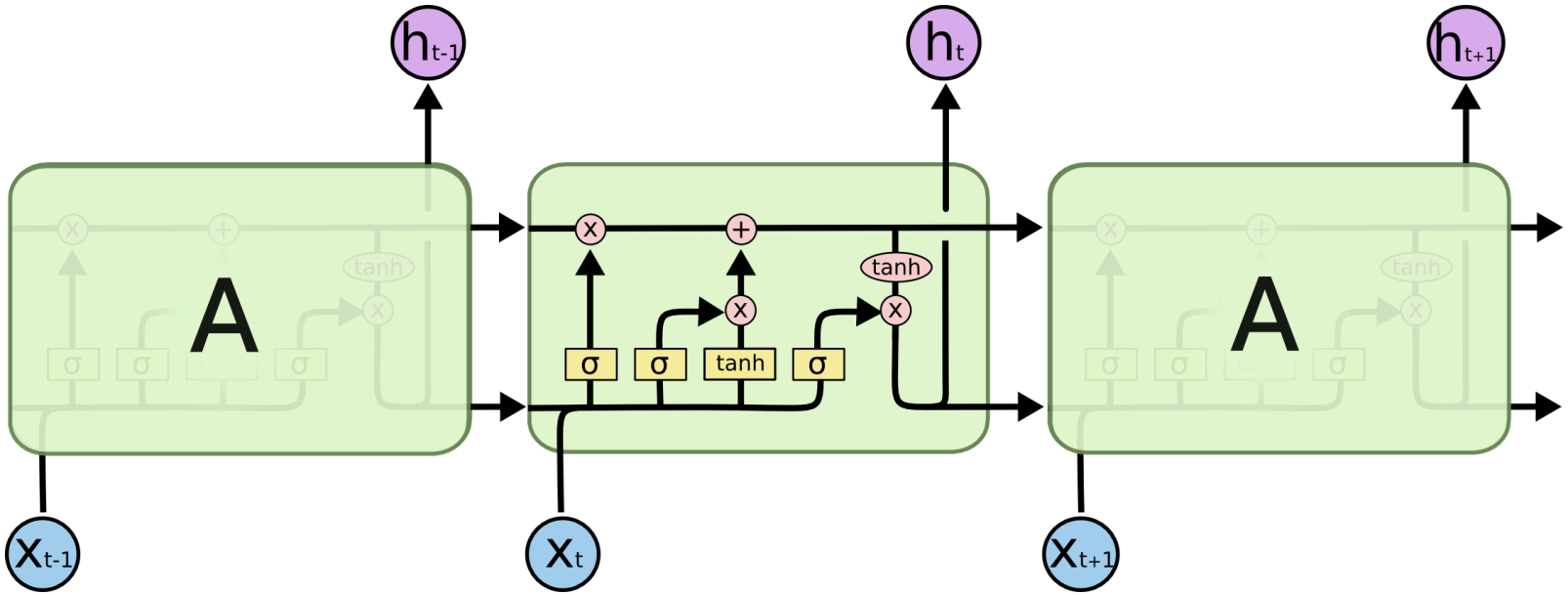
Sepp Hochreiter, Jürgen Schmidhuber

Technische Universität München, IDSIA · Neural Computation 1997

Hochreiter & Schmidhuber (1997): a recurrent cell that can keep information for a long time.

LSTM, Long Short-Term Memory

The **long short-term memory (LSTM)** network is a recurrent architecture composed of a **cell state**, an **input gate**, an **output gate**, and a **forget gate**.



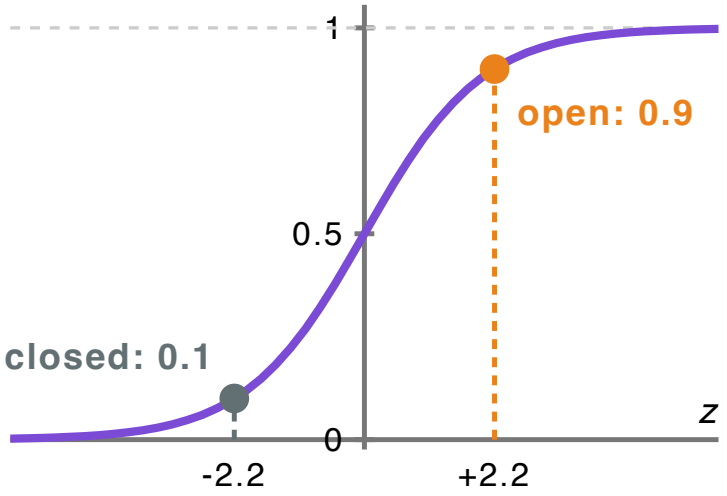
The cell state can remember values **over long time intervals**, and the three gates regulate the flow of information into and out of the cell.

Gates

Gates are composed of a **sigmoid layer** and a **pointwise multiplication** operation (Hadamard product).

$$\sigma\left(\begin{bmatrix}2.2\\-2.2\end{bmatrix}\right) \odot \begin{bmatrix}5\\8\end{bmatrix} = \begin{bmatrix}0.9\\0.1\end{bmatrix} \odot \begin{bmatrix}5\\8\end{bmatrix} = \begin{bmatrix}4.5\\0.8\end{bmatrix}$$

The first value passes almost unchanged, the second is almost blocked.



Gates control how much information enters or leaves the cell state.

The intuition is:

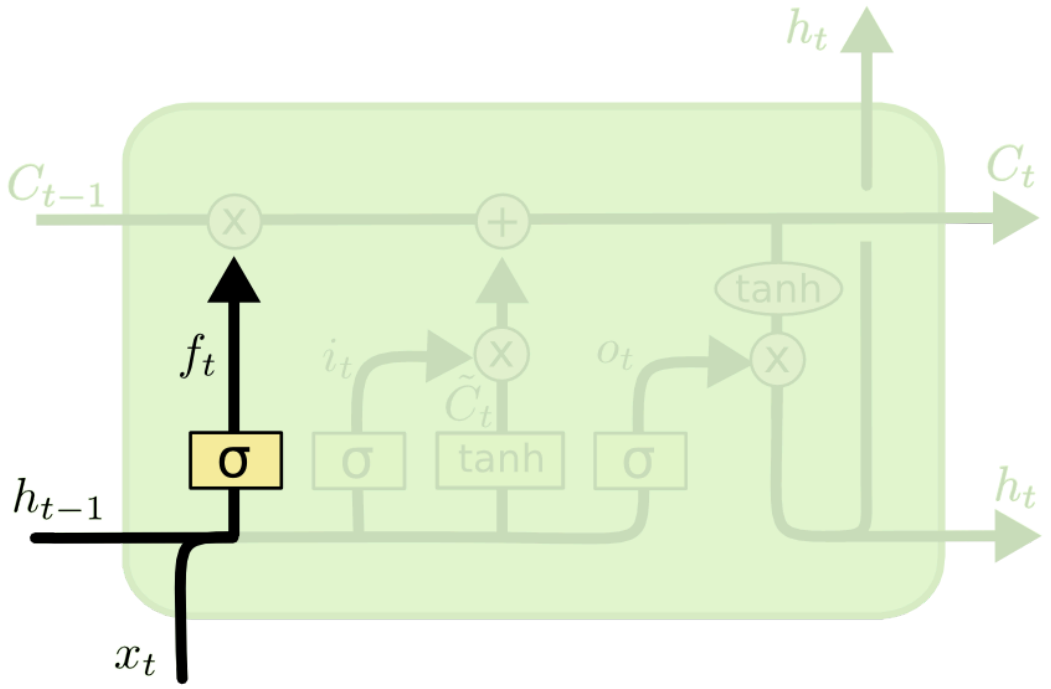
- multiply by something close to 0 $\rightarrow$ gate closed
- multiply by something close to 1 $\rightarrow$ gate open

Forget Gate

The **forget gate** decides which parts of the previous cell state $\mathbf{C}_{t-1}$ to erase, by looking at the previous state $\mathbf{h}_{t-1}$ and the current word $\mathbf{x}_t$. Its values are between 0 (forget) and 1 (keep).

$$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t])$$

$[\mathbf{h}_{t-1}, \mathbf{x}_t]$ is the concatenation of the two vectors, and σ the sigmoid.



Input Gate

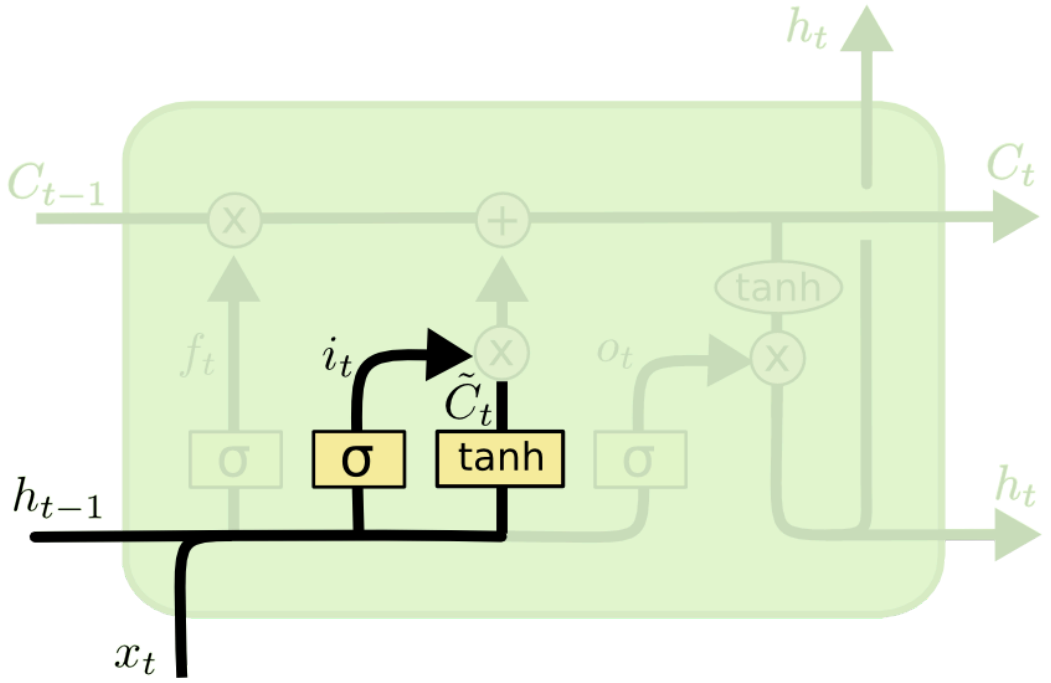
Two parts decide what **new information** to store in the cell state:

- an **input gate**, with values between 0 and 1, that says how much of each new value to let in,

$$\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t])$$

- and a **tanh layer** that creates the new candidate values $\tilde{\mathbf{C}}_t$.

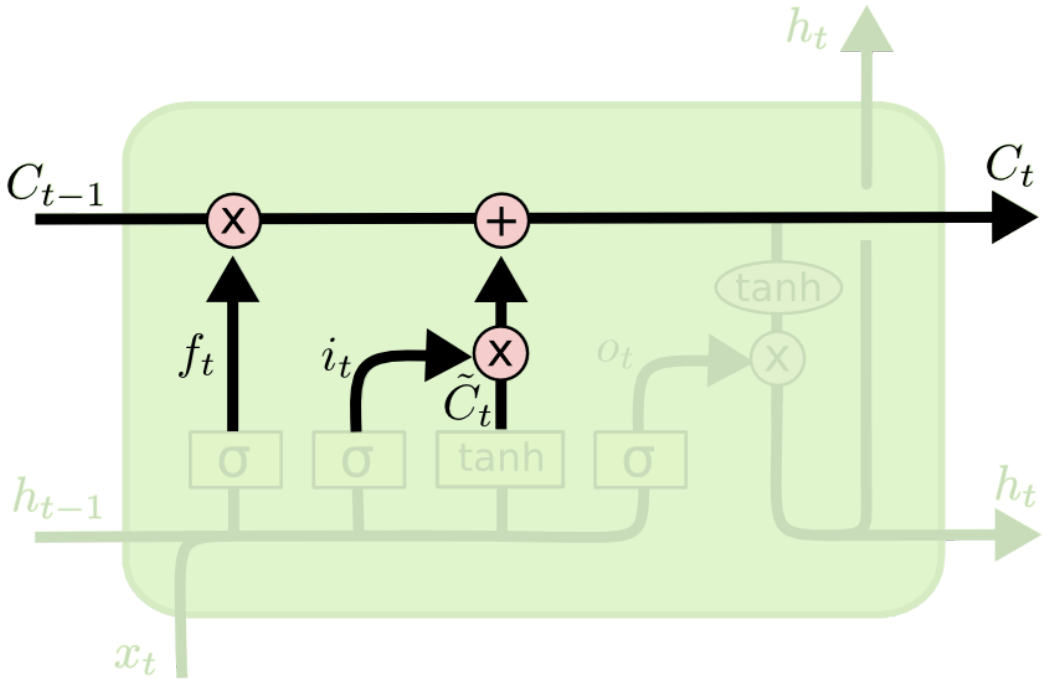
$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{W}_C[\mathbf{h}_{t-1}, \mathbf{x}_t])$$



Cell State Update

The **old cell state $\mathbf{C}_{t-1}$ is updated** into the cell state $\mathbf{C}_t$ by forgetting and adding new information:

$$\mathbf{C}_t = \mathbf{f}_t \odot \mathbf{C}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{C}}_t$$

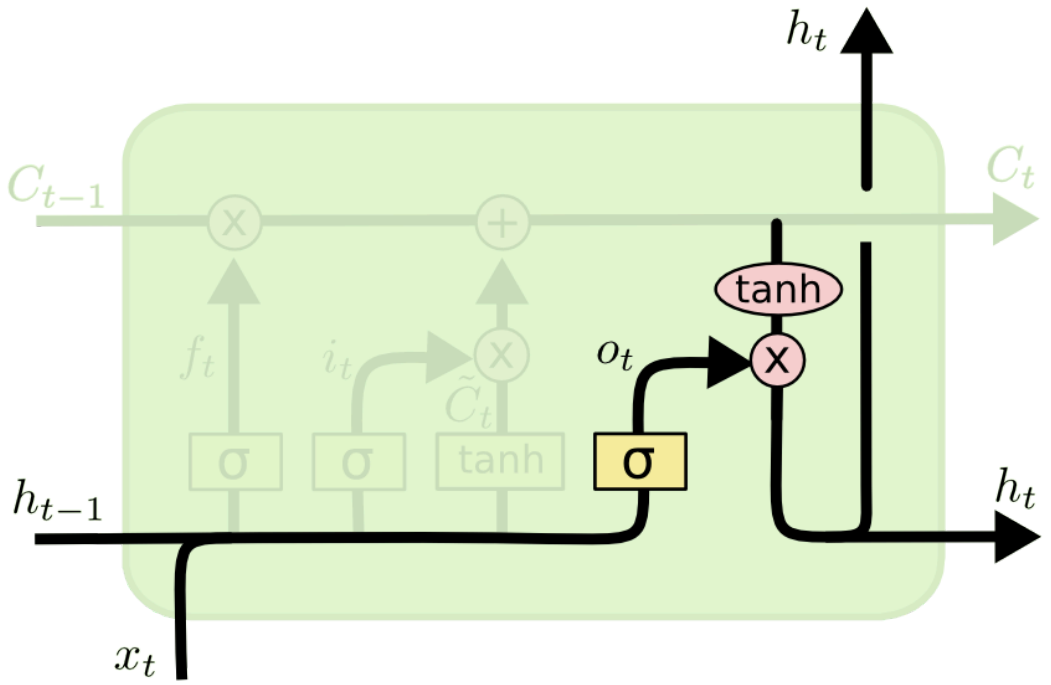


Output Gate

The **output gate** decides which parts of the cell state are written in the hidden state $\mathbf{h}_t$, the output of the step. The cell state goes through a pointwise $\tanh$ and is gated:

$$\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t])$$

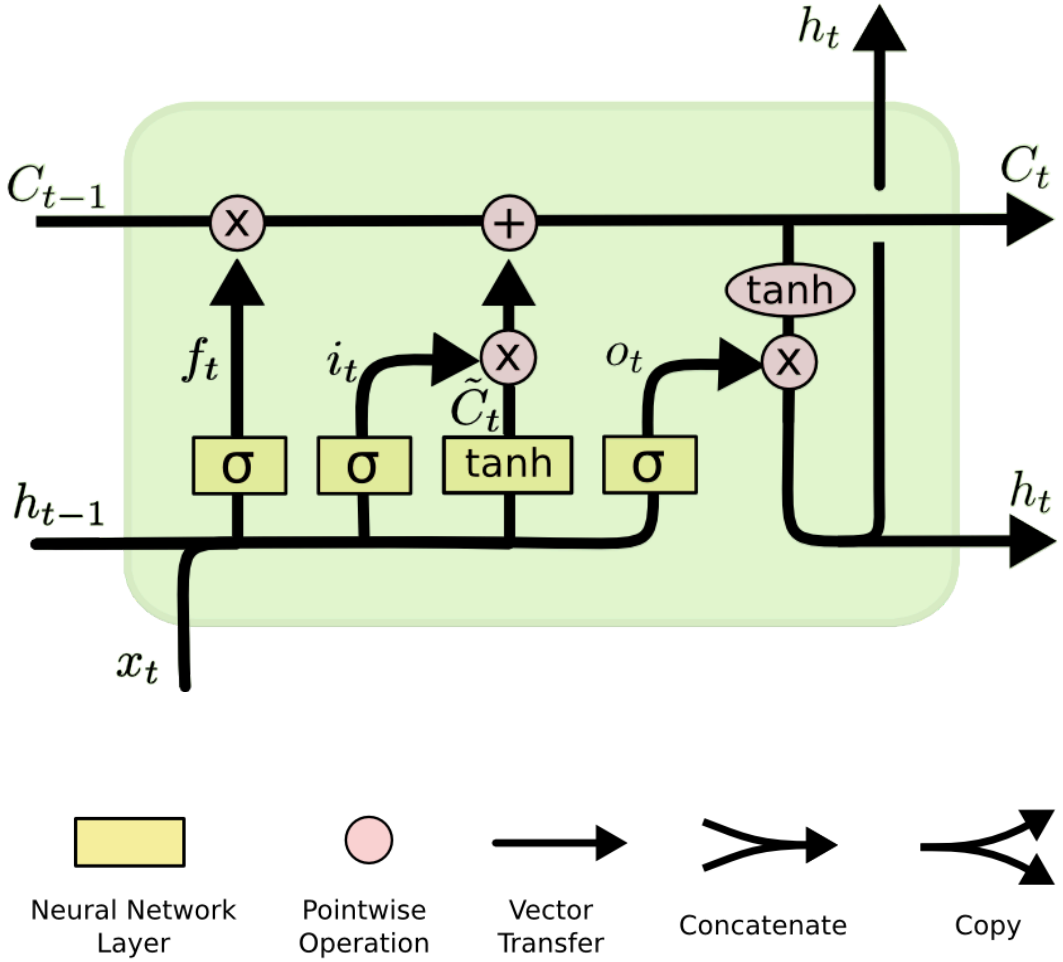
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{C}_t)$$



LSTM Summary

One step of the LSTM, from the previous states $\mathbf{h}_{t-1}$, $\mathbf{C}_{t-1}$ and the new word $\mathbf{x}_t$:

$$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t])$$
$$\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t])$$
$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{W}_C[\mathbf{h}_{t-1}, \mathbf{x}_t])$$
$$\mathbf{C}_t = \mathbf{f}_t \odot \mathbf{C}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{C}}_t$$
$$\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t])$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{C}_t)$$



Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation

Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, Yoshua Bengio

Université de Montréal, Jacobs University, Université du Maine · EMNLP 2014

Cho et al. (2014): a simpler gated cell, introduced for machine translation.

GRU, Gated Recurrent Unit

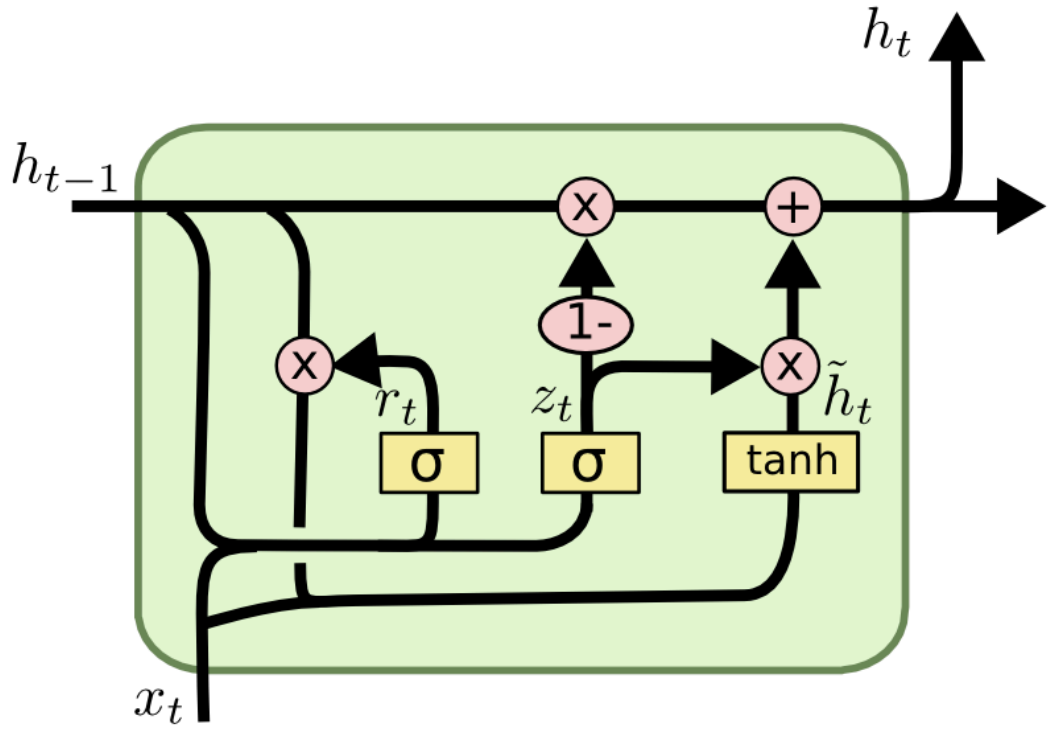
The **Gated Recurrent Unit (GRU)** uses an **update gate $\mathbf{z}_t$** and a **reset gate $\mathbf{r}_t$**, and merges the cell and hidden states.

$$\mathbf{z}_t = \sigma(\mathbf{W}_z[\mathbf{h}_{t-1}, \mathbf{x}_t])$$

$$\mathbf{r}_t = \sigma(\mathbf{W}_r[\mathbf{h}_{t-1}, \mathbf{x}_t])$$

$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}[\mathbf{r}_t \odot \mathbf{h}_{t-1}, \mathbf{x}_t])$$

$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t$$



- **Update gate**: how much of the state to update with the new candidate.
- **Reset gate**: how much of it to use for the candidate.

Why Gating Helps

The cell state is updated with only a **product** and a **sum**, so when the forget gate is close to 1 the information **and the gradient** pass through almost unchanged, however many steps there are. The network **learns** when to open and close the gates.

In *"The cats that the dog chased all day long **are** tired"*, a unit can remember the plural of *cats* through the whole clause, and use it at *are*.

Deep Contextualized Word Representations

Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, Luke Zettlemoyer

Allen Institute for AI, University of Washington · NAACL 2018

Peters et al. (2018): word vectors that depend on the sentence.

ELMo, Contextual Embeddings

ELMo (Embeddings from Language Models) is a **bidirectional LSTM** trained as a **language model** on a large corpus. At every word, its hidden states become the vector of that word.

$$\underbrace{\mathbf{x}_t}_{\text{non-contextual input}} \xrightarrow{\text{BiLSTM}} \underbrace{\mathbf{h}_t = [\overrightarrow{\mathbf{h}_t}; \overleftarrow{\mathbf{h}}_t]}_{\text{contextual representation}}$$

These vectors are **contextual embeddings**: they depend on the whole sentence. In *"river **bank**"* and *"investment **bank**"* the word has the same static embedding, but different ELMo vectors.

They are then used as the input of other tasks, such as classification or tagging.



Where RNNs Stand

Most text tasks today don't use RNNs, but they are still useful when:

- text arrives as a **stream** and the prediction must be **updated at every word**
- the model must be **small and fast**, with constant memory

Their limits: they read one token at a time, so the **steps cannot run in parallel**; the whole past is squeezed into a **fixed-size** vector; **long dependencies** are still hard, even with gates.

The next step is to let the model look **directly** at all the tokens, with **attention**.

Sources

Textbooks

- Jurafsky & Martin, *Speech and Language Processing*, 3rd ed. (Jan 2026 draft), ch. 13, *RNNs and LSTMs*

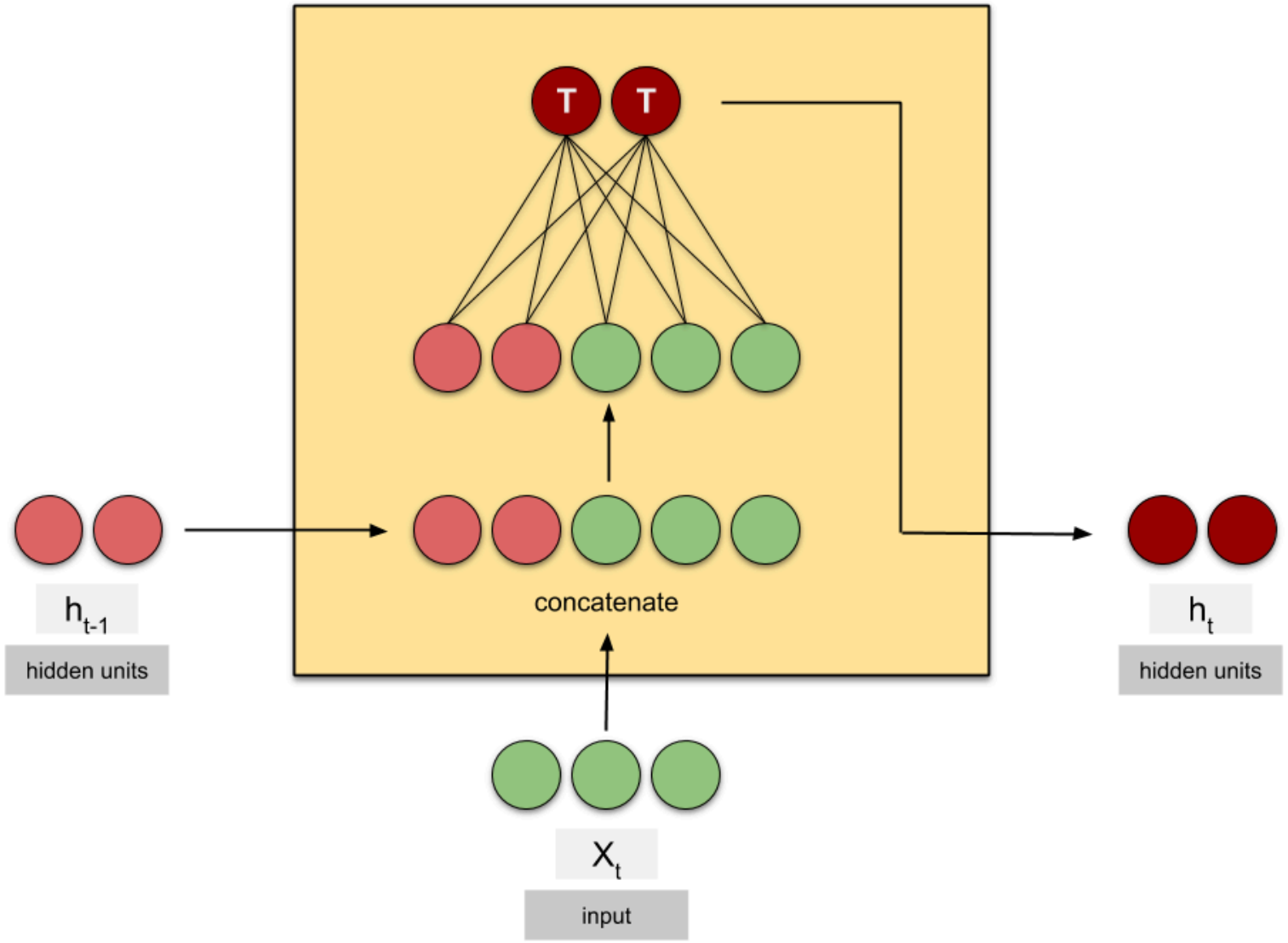
Papers

- Elman (1990), *Finding Structure in Time*.
- Hochreiter & Schmidhuber (1997), *Long Short-Term Memory*.
- Pascanu, Mikolov & Bengio (2013), *On the Difficulty of Training Recurrent Neural Networks*.
- Cho et al. (2014), *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*.
- Peters et al. (2018), *Deep Contextualized Word Representations* (ELMo).

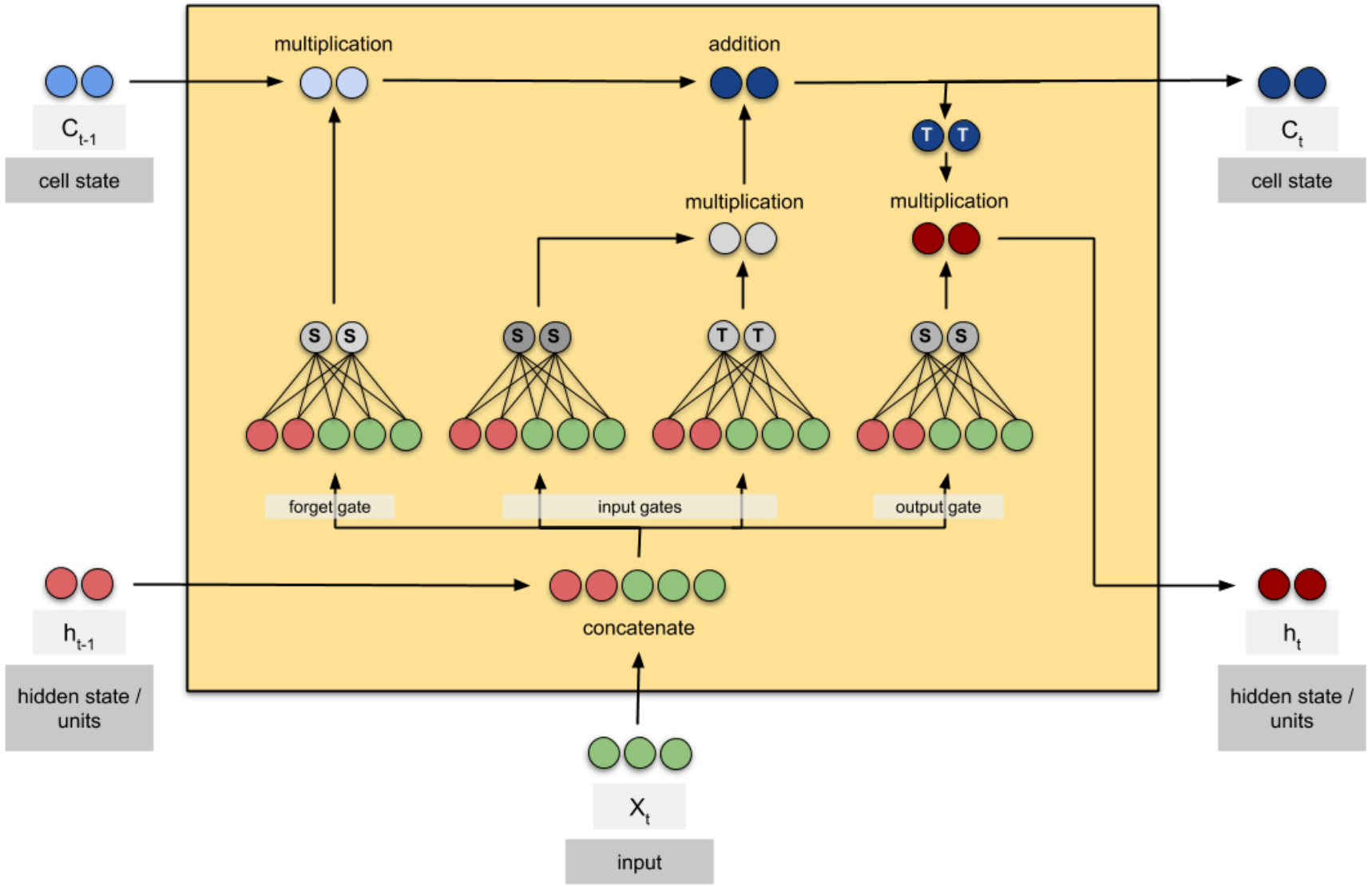
Appendix

The cells in detail

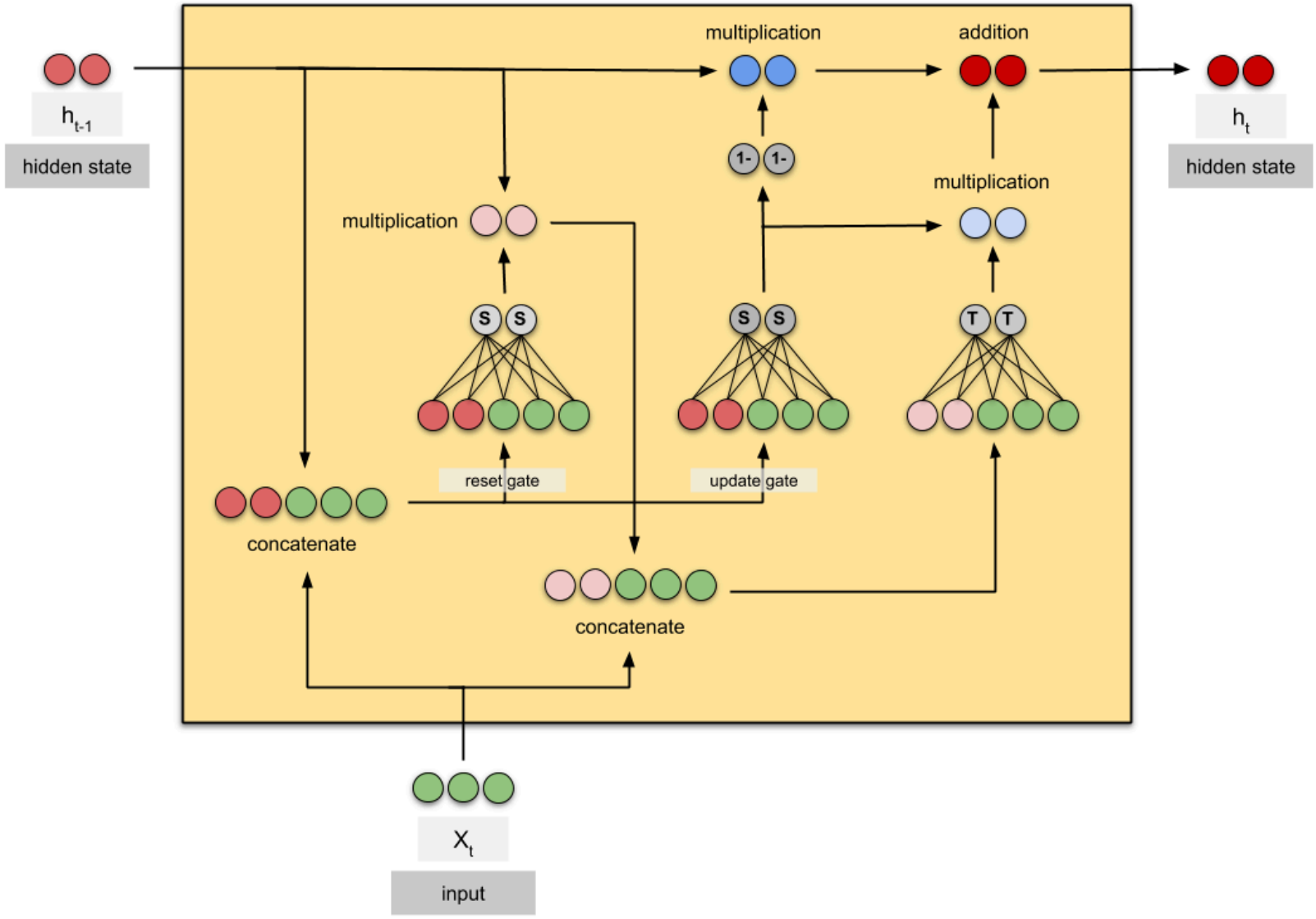
RNN Detail



LSTM Detail


multiplication
addition
C_{t-1}
cell state
C_t
cell state
T T
multiplication
multiplication
S S
S S
T T
S S
forget gate
input gates
output gate
h_{t-1}
hidden state / units
concatenate
h_t
hidden state / units
X_t
input

GRU Detail


multiplication
addition
h_{t-1}
hidden state
1-
1-
multiplication
multiplication
h_t
hidden state
S
S
S
S
T
T
reset gate
update gate
concatenate
concatenate
X_t
input