ARTIFICIAL INTELLIGENCE 2

Recurrent Neural Networks

Francesco Spinnato

* francesco.spinnato@unipi.it
University of Pisa

Recurrent Neural Networks

Sequential Data

Dealing with Sequential Data

SEQUENTIAL DATA: any kind of data where the order of the observations matters.

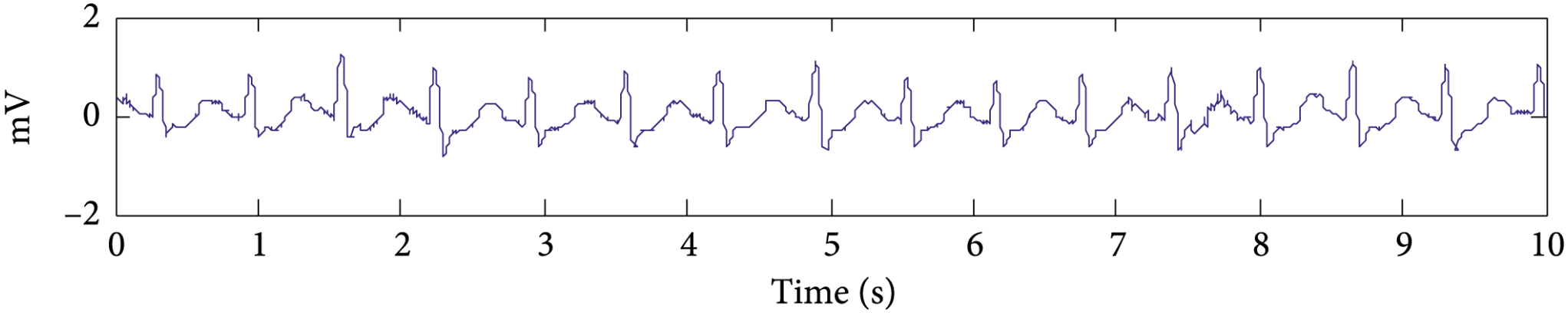
The sole importance of the crossing of the Berezina lies in the fact that it plainly and indubitably proved the fallacy of all the plans for cutting off the enemy's retreat and the soundness of the only possible line of action--the one Kutuzov and the general mass of the army demanded--namely, simply to follow the enemy up. The French crowd fled at a continually increasing speed and all its energy was directed to reaching its goal. It fled like a wounded animal and it was impossible to block its path. This was shown not so much by the arrangements it made for crossing as by what took place at the bridges. When the bridges broke down, unarmed soldiers, people from Moscow and women with children who were with the French transport, all--carried on by vis inertiae--pressed forward into boats and into the ice-covered water and did not, surrender.

Time Series

Time series channel: $\mathbf{x} = [x_1, x_2, \ldots, x_t] \in \mathbb{R}^t$, ordered real-valued observations. We assume regular sampling, i.e., Δt is constant.

Time series: $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_d\} \in \mathbb{R}^{d \times t}$, collection of d channels

Time series dataset: $\mathcal{X} = \{\mathbf{X}_1, \ldots, \mathbf{X}_n\} \in \mathbb{R}^{n \times d \times t}$, collection of n time series



Time series are ready-to-go in a neural network (they are just vectors/matrices).

Audio Spectrograms

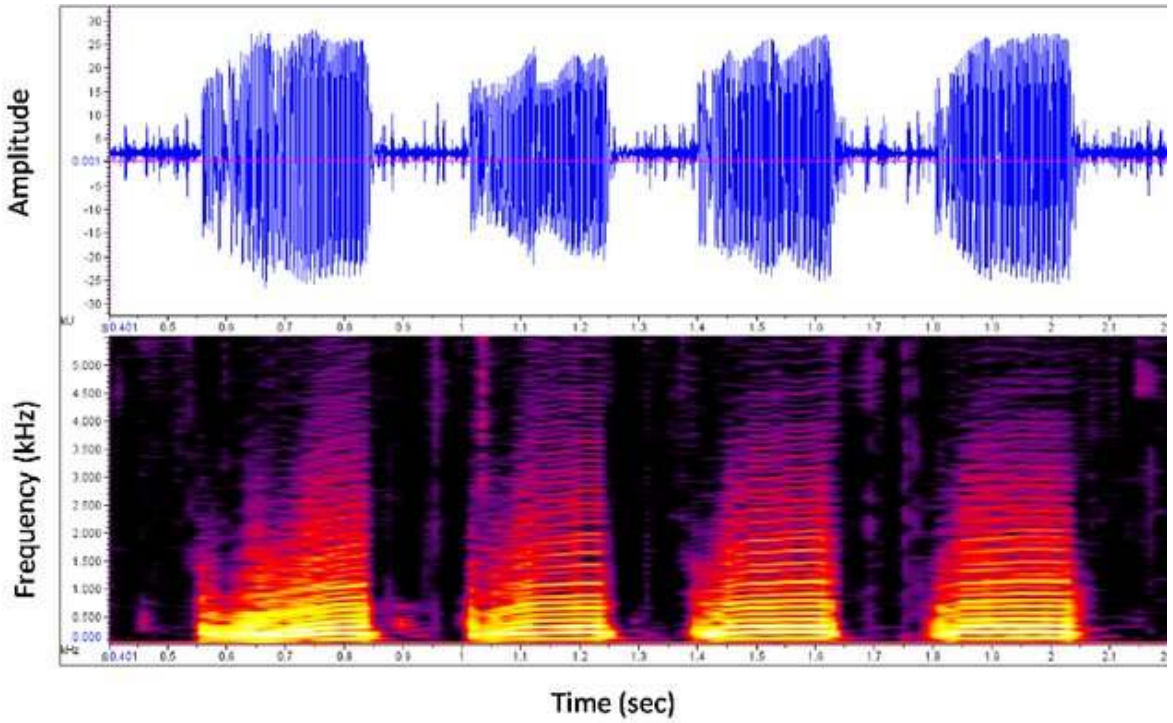
Audio can be represented as a time series channel: $\mathbf{x} = [x_1, x_2, \ldots, x_t] \in \mathbb{R}^t$.

A common alternative is a **spectrogram**, i.e. a time-frequency representation:

Spectrogram: $\mathbf{S} \in \mathbb{R}^{f \times t}$

Each column (time frame) is a feature vector $\mathbf{s}_\tau \in \mathbb{R}^f$, so the spectrogram is naturally a **sequence of vectors**.

(Often used: log-power, mel, or log-mel spectrograms.)



Text

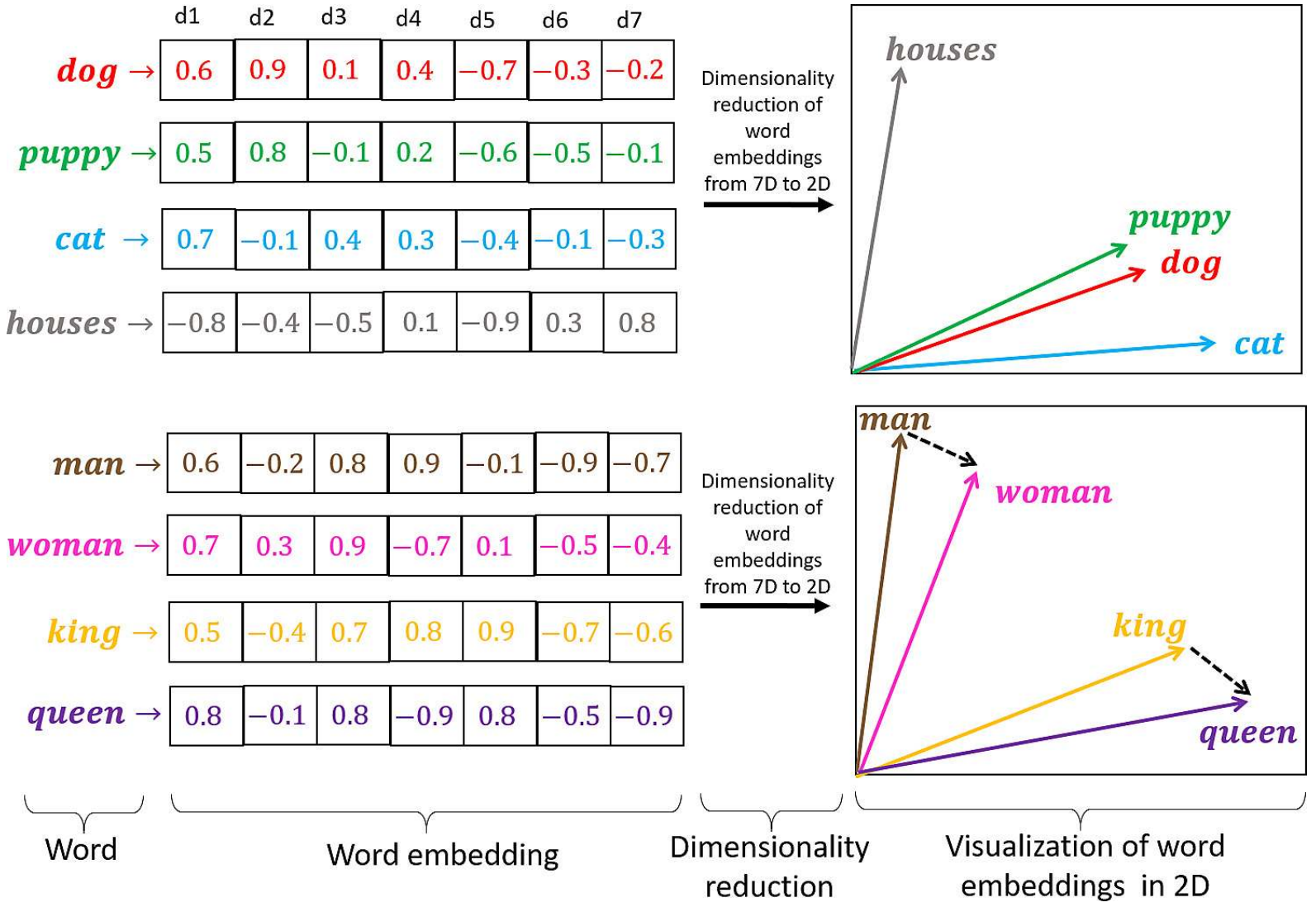
Text is a sequence of characters, words, sentences... But it is not numerical!

We will go into more details in later lectures but in summary, to work with text we first need to **convert it into continuous numerical vectors**. This is done through:

- **Tokenization**: Split text into tokens (words, subwords, or characters)
- **Indexing**: Map each token to an integer ID via a vocabulary.
- **Embedding**: Map integer IDs to dense vectors using
  - an embedding layer
  - pretrained embeddings (e.g., Word2vec)

Text Embeddings

The embedding space is semantically meaningful, i.e., words that are somewhat related are closer



Dealing with Sequential Data

	x - input		y - output
Speech recognition		⟹	"Fuzzy Wuzzy was a bear. Fuzzy Wuzzy had no hair."
Music generation	Ø	⟹	
Sentiment classification	"Decent effort. The plot could have been better."	⟹	
DNA sequence analysis	ACTGTACCCATGTGACTGCCC	⟹	ACTGTACCCATGTGACTGCCC
Machine translation	"El que no arriesga, no gana."	⟹	"If you don't take risks, you cannot win."
Video activity recognition		⟹	Running
Name entity recognition	"Ygritte says Jon Snow knows nothing."	⟹	"Ygritte says Jon Snow knows nothing."

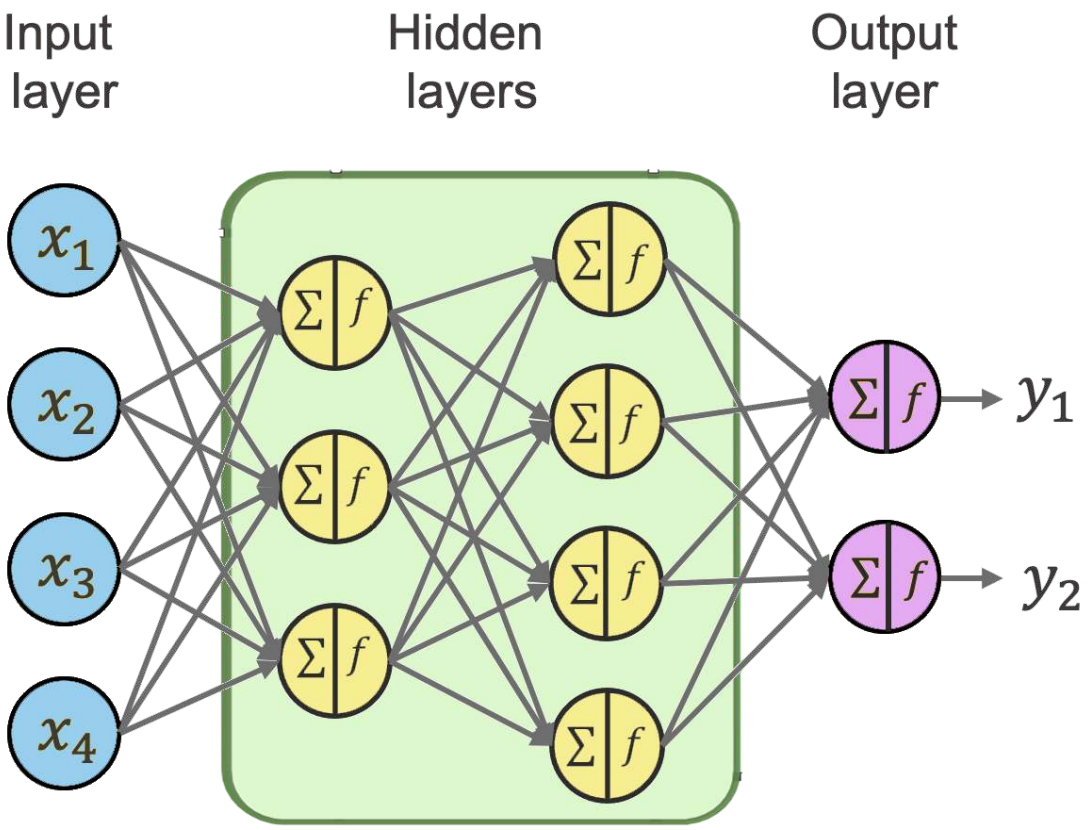
Why not vanilla neural networks?

The key parts of a vanilla neural network are: the **input**, the **output**, the **hidden layers**, and the **weights** between the nodes.

Vanilla Neural Networks take a fixed-sized input and produce a fixed-sized output.

$$\mathbf{h} = f_{\mathbf{w}_{xh}}(\mathbf{x})$$

$$\hat{\mathbf{y}} = f_{\mathbf{w}_{hy}}(\mathbf{h})$$



Why not vanilla neural networks?

Vanilla neural networks have some problems in dealing with sequential data:

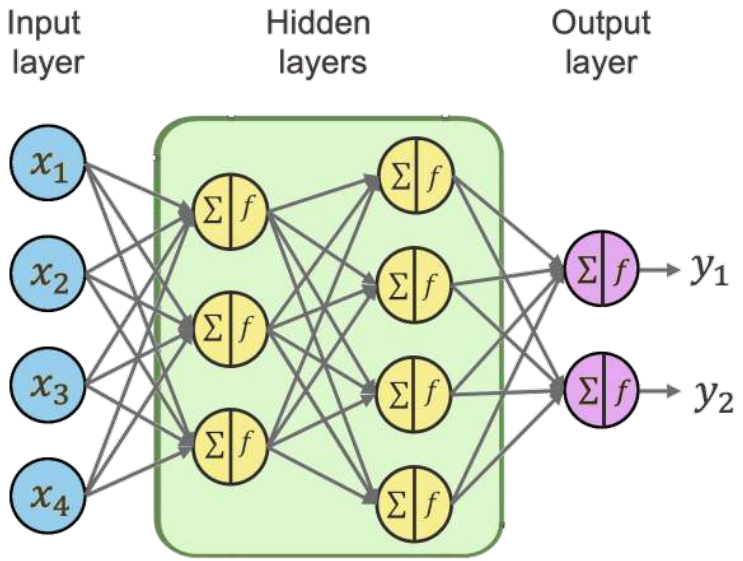
- they **don't share parameters**, i.e. can't learn similar features in different parts of the sequential data;
- they can't handle **different sized data**;
- they **don't consider the sequential structure of the data**: data points depend upon previous data points.

Why not vanilla neural networks?

Yesterday I ate pizza

I ate pizza yesterday

What did you eat yesterday?



Recurrent Neural Networks

Vanilla RNNs

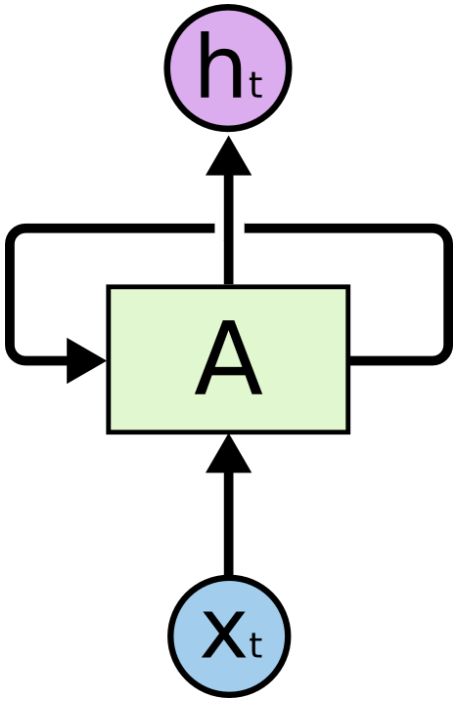
RNN

Recurrent neural networks (RNNs) are a class of neural networks that allow **previous outputs to be used as inputs** of the network.

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

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

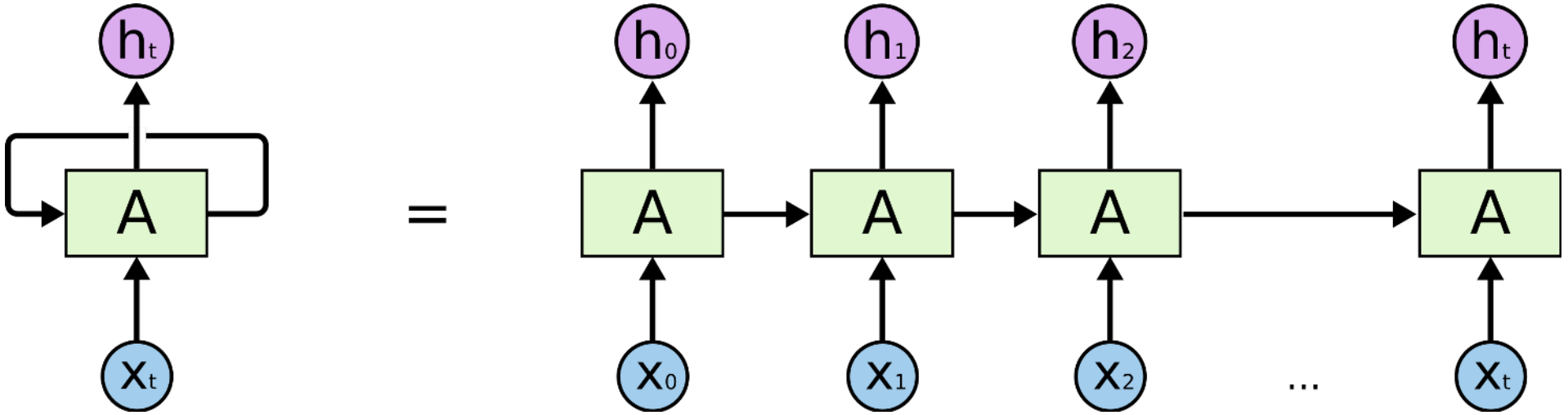
- $\mathbf{x}_t$: input at time-step t
- A: memory cell
- $\mathbf{h}_t$: new state
- $\mathbf{h}_{t-1}$: old state
- $f_{\mathbf{W}}$: some activation function with parameters $\mathbf{W}$



L. C. Jain and L. R. Medsker, Recurrent Neural Networks: Design and Applications, 1999.

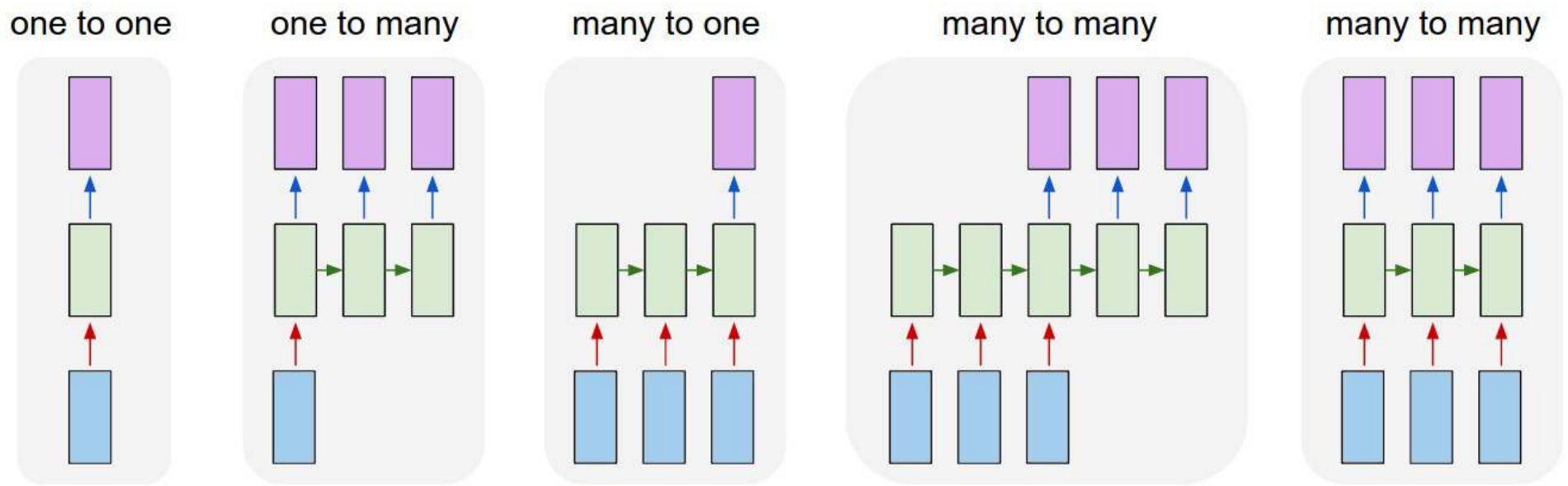
RNN

The RNN can be thought of as **multiple copies of the same network**, each passing a message to a successor.



Different Applications

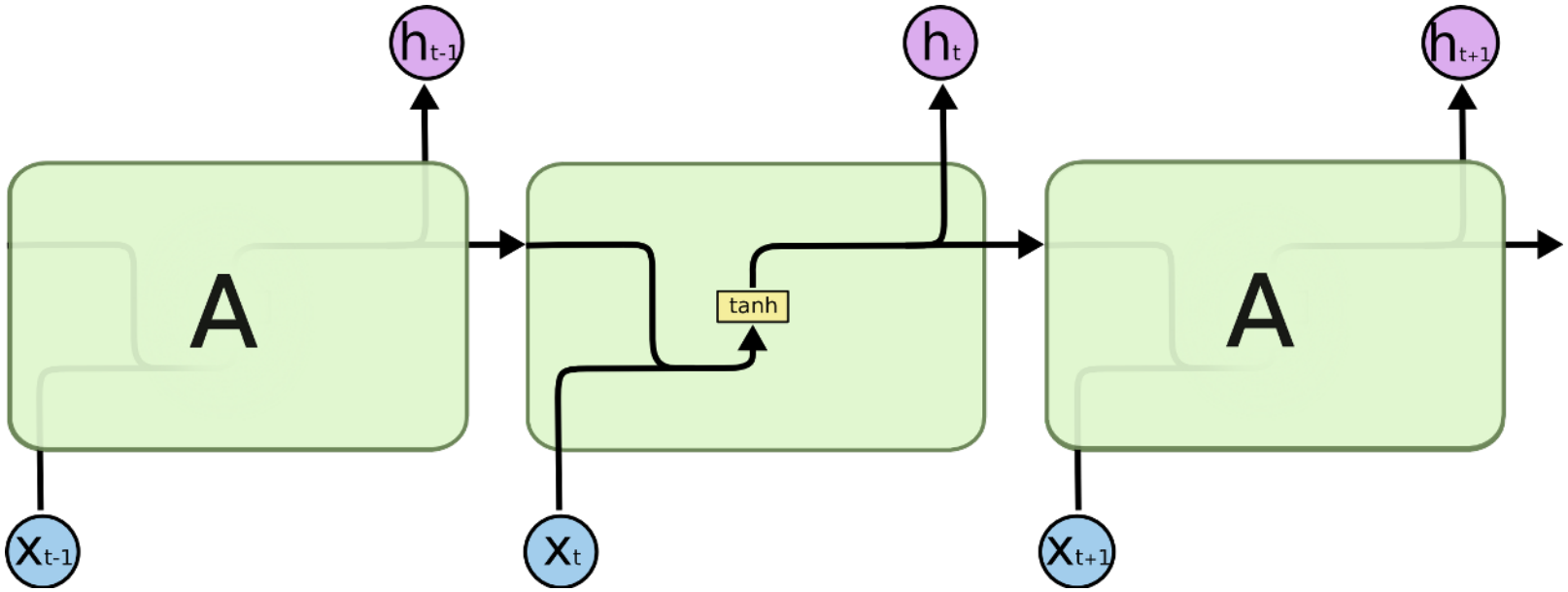
There are different kinds of possible network structures depending on the task.

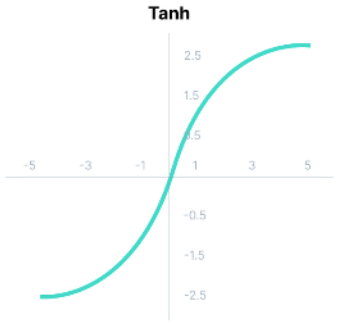


- **one to one**: vanilla neural network
- **one to many**: e.g. image captioning (image → sequence of words)
- **many to one**: e.g. sentiment classification (sequence of words → sentiment)
- **many to many**: e.g. machine translation (sequence of words → sequence of words)

Vanilla RNN

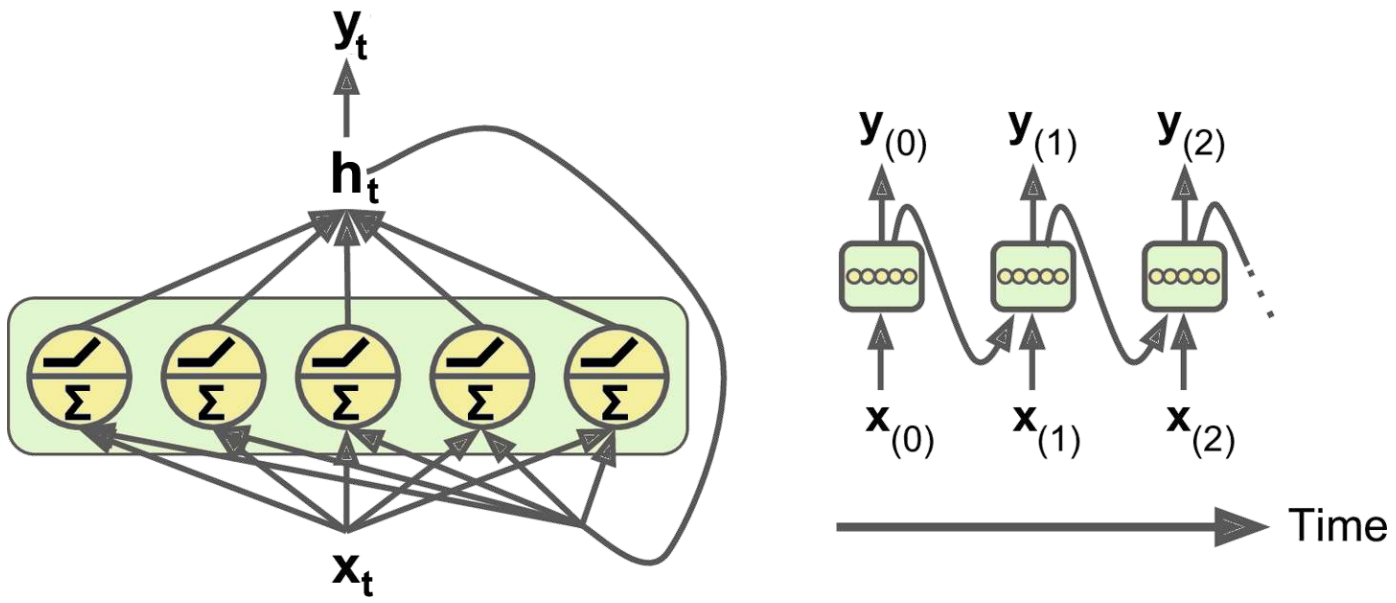
$$h_t = \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t)$$





Output

You can have **multiple neurons** inside the memory cell.



The output of an RNN is:

$$\hat{\mathbf{y}}_t = \mathbf{W}_{hy}\mathbf{h}_t$$

Exercise: RNN Forward

We consider a simple RNN with **2 inputs** and **1 hidden unit.**

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

The hidden state update is:

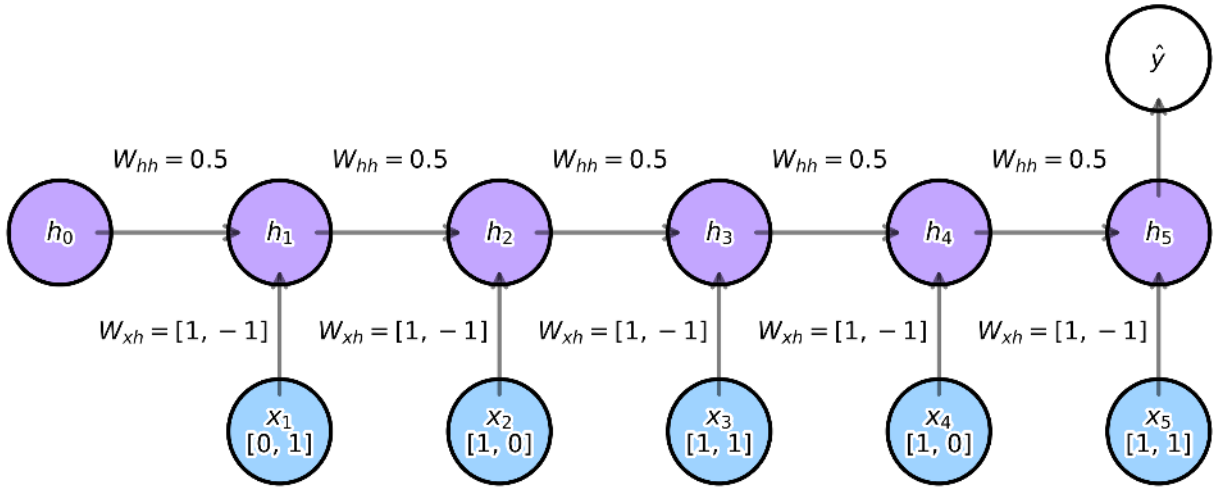
$$h_0 = 0, \quad h_t = \mathrm{ReLU}\left(0.5\, h_{t-1} + \mathbf{x}_t \begin{bmatrix} 1 \\ -1 \end{bmatrix}\right)$$

Our sequence is:

$$\mathbf{x}_1 = [0 \quad 1], \quad \mathbf{x}_2 = [1 \quad 0], \quad \mathbf{x}_3 = [1 \quad 1], \quad \mathbf{x}_4 = [1 \quad 0], \quad \mathbf{x}_5 = [1 \quad 1]$$

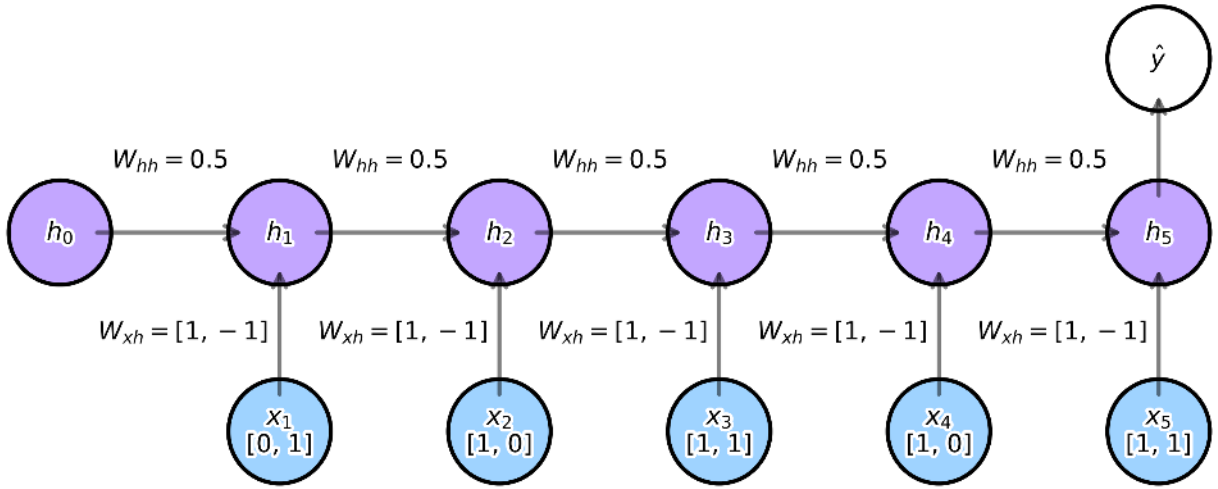
Exercise: RNN Forward $(t = 1)$

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



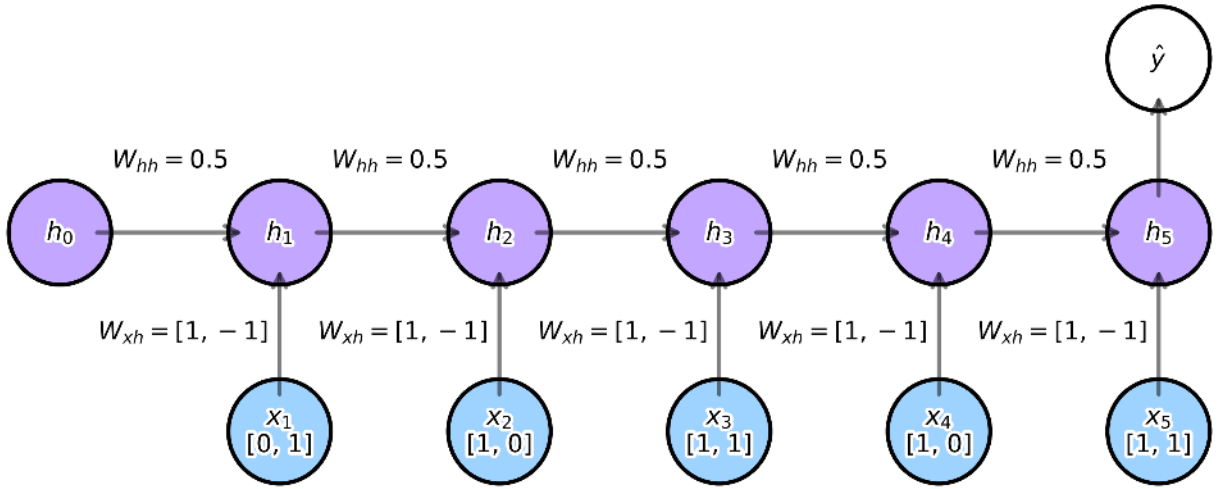
Exercise: RNN Forward $(t = 2)$

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



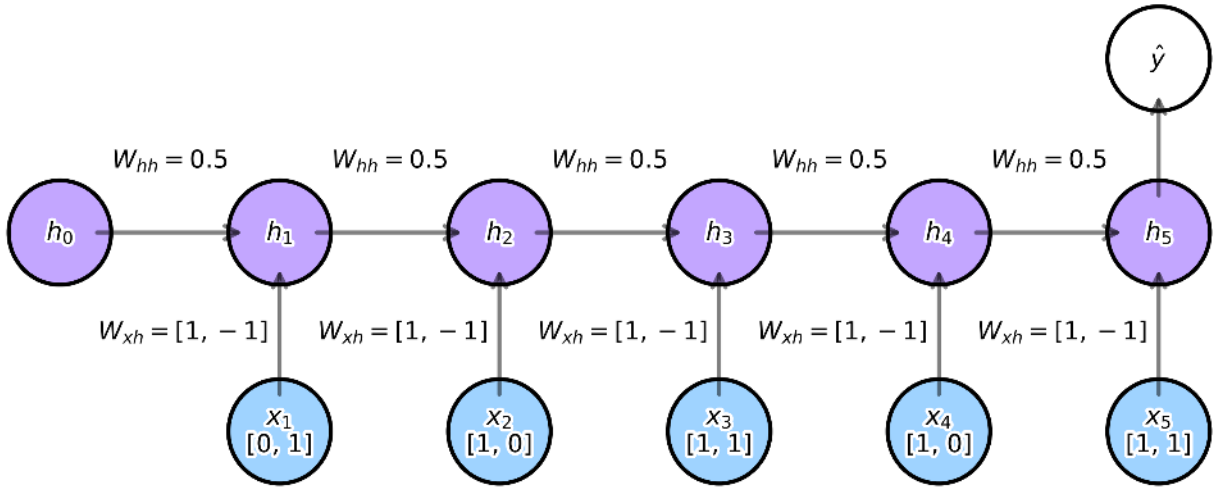
Exercise: RNN Forward $(t = 3)$

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



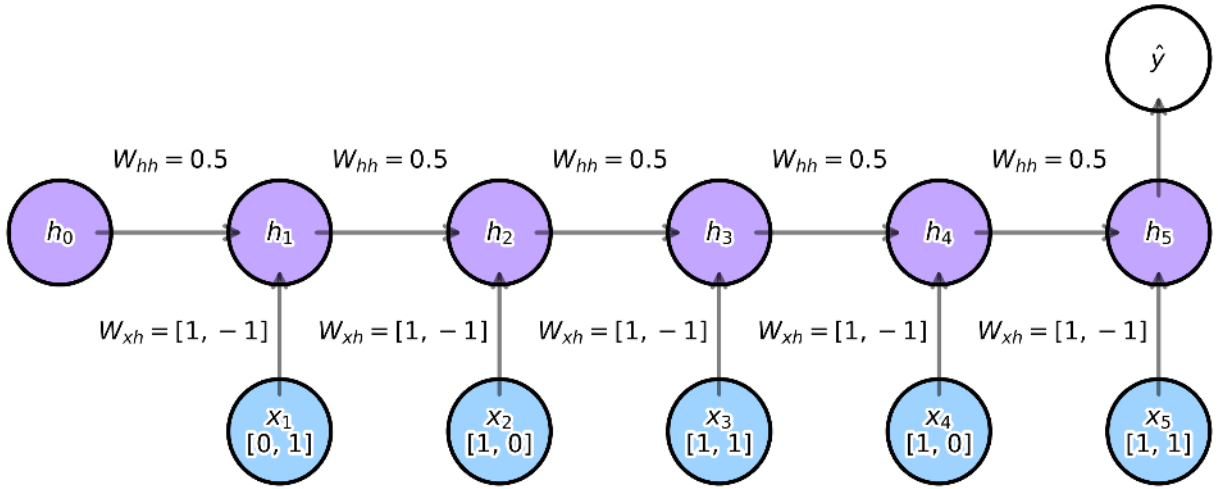
Exercise: RNN Forward $(t = 4)$

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



Exercise: RNN Forward $(t = 5)$

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



Exercise: RNN Forward Output

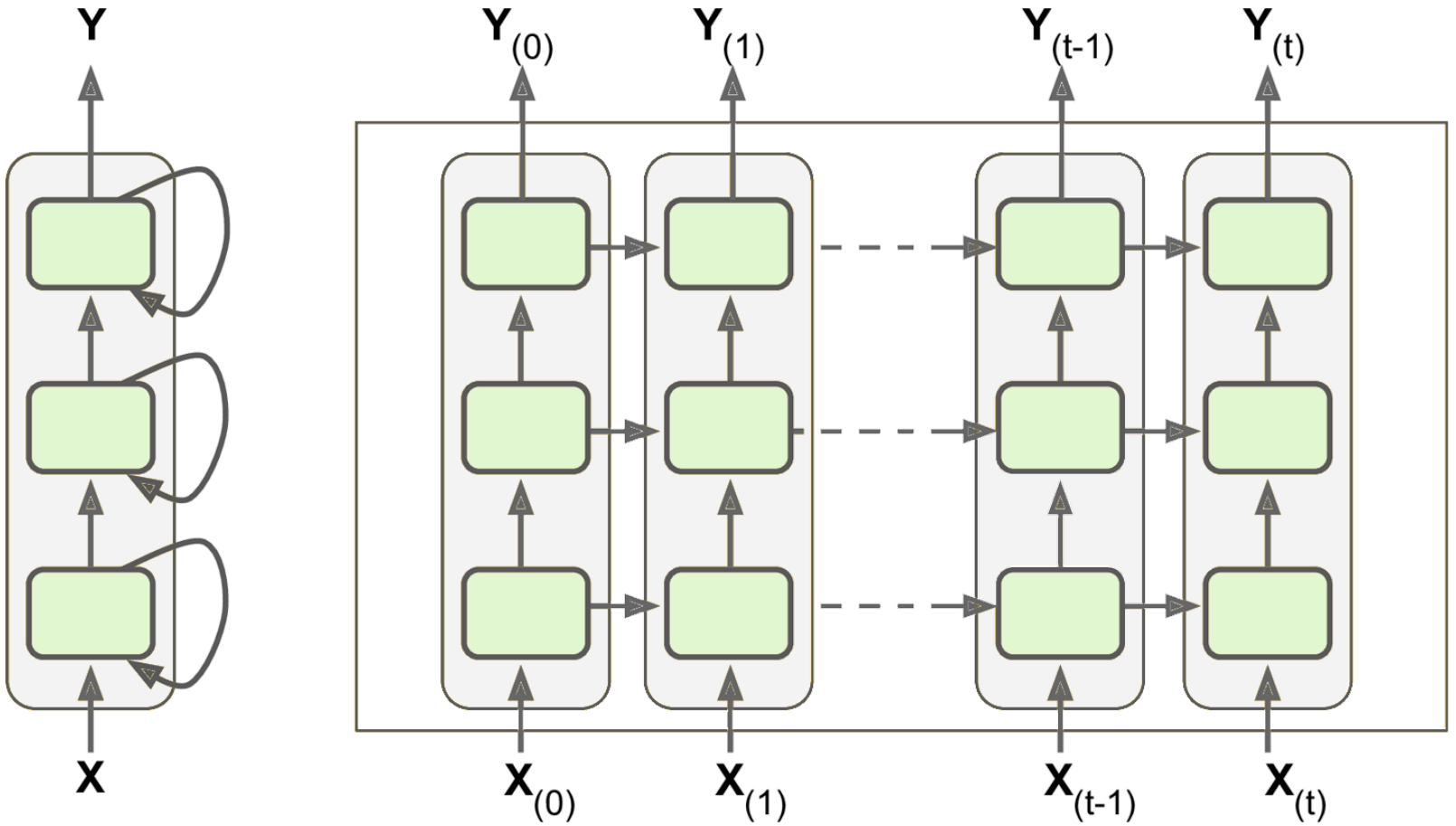
If our RNN is a many-to-one network, the last hidden state is used to compute the final label, e.g. with a sign activation function like in the original perceptron.

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

We just applied the forward in a **recurrent perceptron**.

Deep RNNs

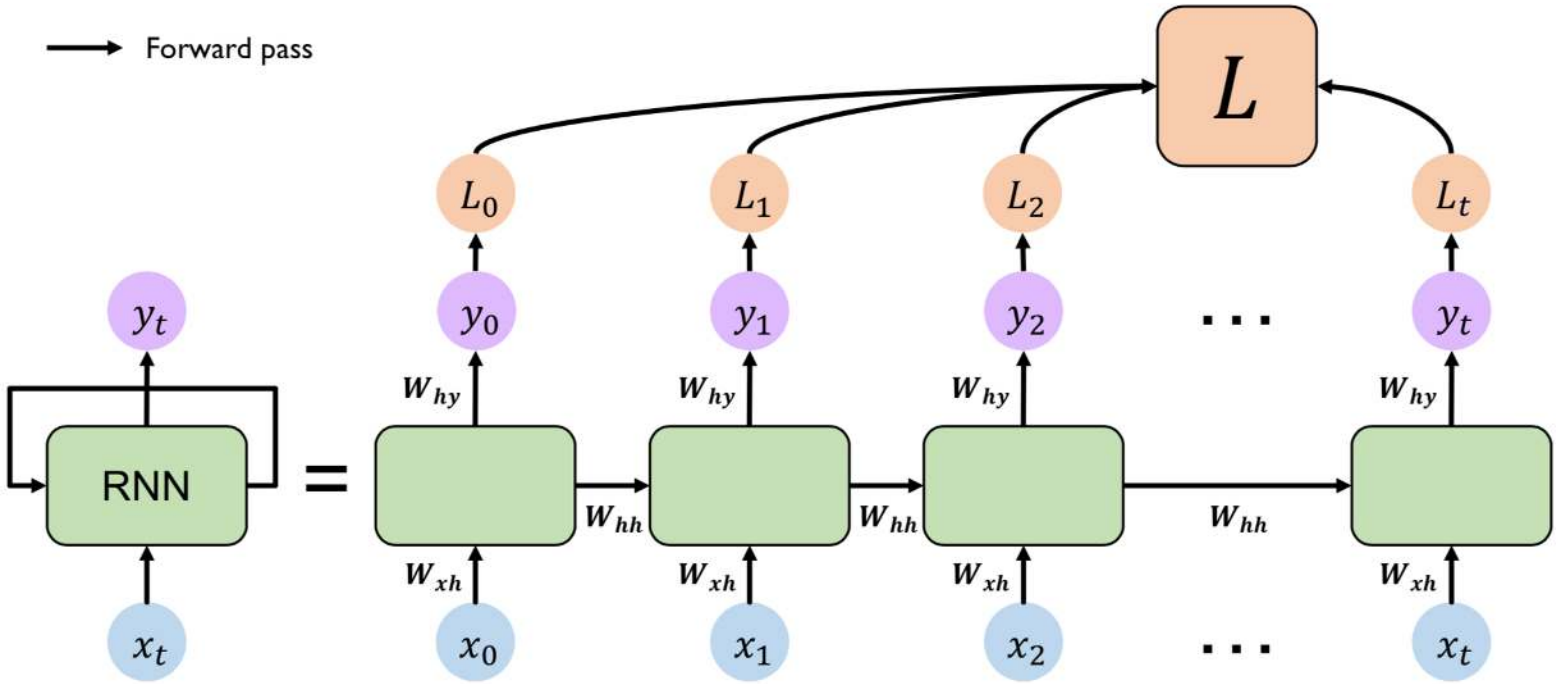
A **deep RNN** can be built by stacking multiple memory cells.



Training: Forward Pass

In the **forward pass**, the output $\hat{y}$ is compared to the ground truth y.

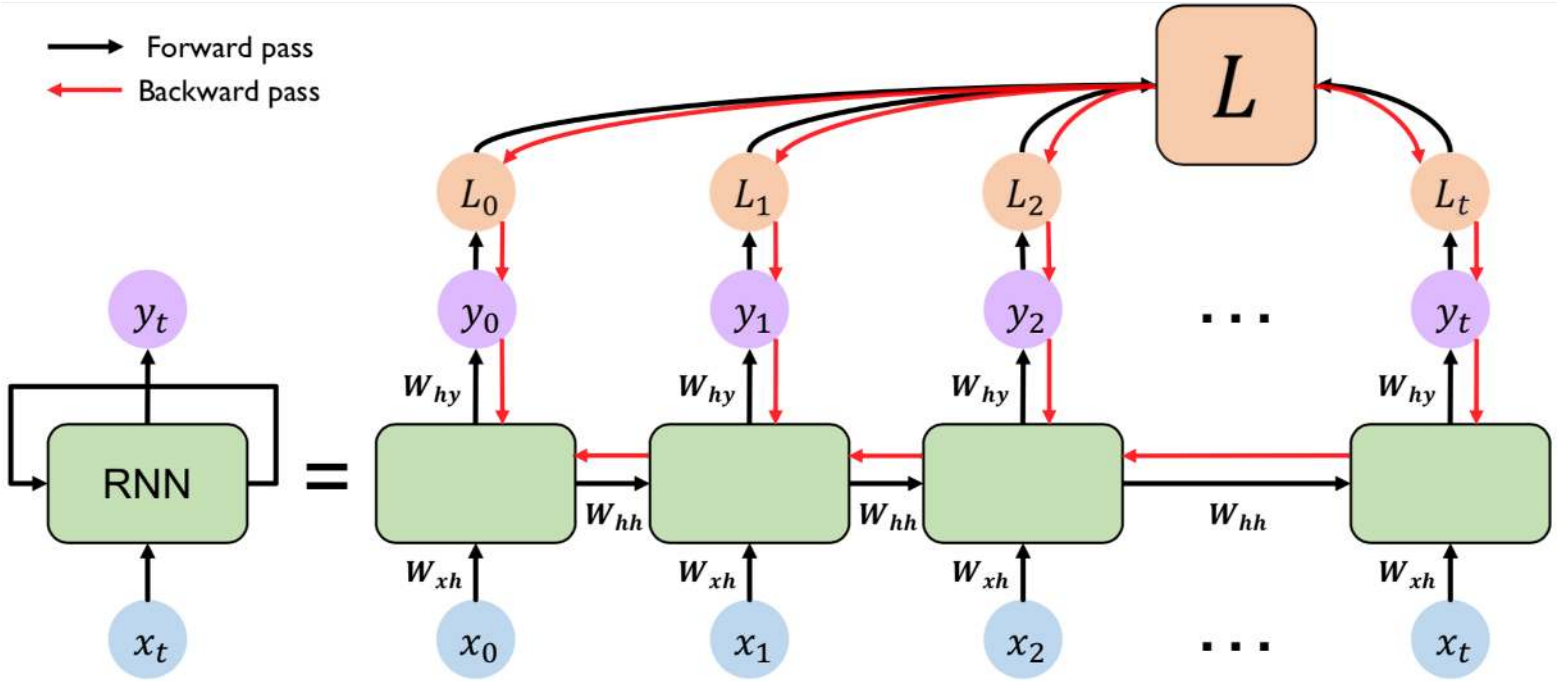
$$\mathcal{L}(y, \hat{y}) = \sum_{t=0}^{T} \mathcal{L}_t(y_t, \hat{y}_t)$$



Training: Backward Pass

RNNs are trained using **backpropagation through time.**

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

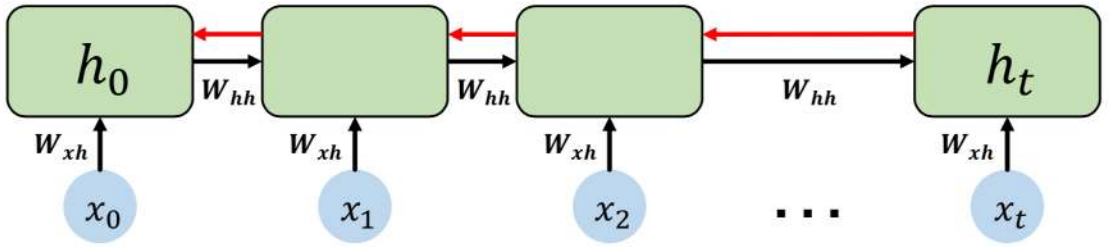


Gradients flow

Given that the weights in the RNN are shared, backpropagating the gradients involves **many matrix multiplications** with $\mathbf{W}$. The gradient is a recursive product of hidden activation gradients.

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} = \prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_{j+1}}{\partial \mathbf{h}_j} = \prod_{j=k+1}^{t} \mathbf{W}^T \times diag\left[f'(\mathbf{h}_{j-1})\right]$$

$$\left\| \frac{\partial \mathbf{h}_{j+1}}{\partial \mathbf{h}_j} \right\| < 1 \quad \text{vanishing gradients} \qquad \left\| \frac{\partial \mathbf{h}_{j+1}}{\partial \mathbf{h}_j} \right\| > 1 \quad \text{exploding gradients}$$

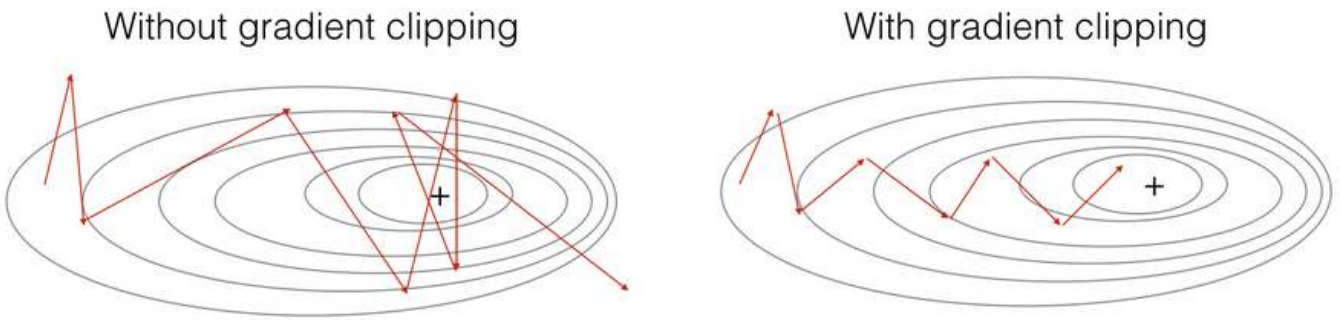


Exploding Gradients

Exploding gradients are a problem because if the gradient value grows extremely large, it causes an **overflow** (i.e. NaN values). They are easily detectable at runtime and can be solved by using **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, gradients are scaled back to a small number whenever they reach a certain threshold, without changing their direction.



Vanishing Gradients

When the gradient value goes to zero, it becomes increasingly hard to update the network weights, reducing the learning quality of the model for far-away observations in the input data.

$$\mathbf{W}_{\text{new}} = \mathbf{W}_{\text{old}} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}},$$

Possible solutions are:

- smart **weights initialization** (identity matrix);
- use **Rectified Linear Units (ReLU)** instead of sigmoid or tanh functions (the derivative of ReLU is either 0 or 1);
- **truncated backpropagation**: truncating the time steps at a computationally convenient window size;
- **gating**.

Past and Future Context

A standard RNN builds its representation at time t using only the **past context** $(x_1, \ldots, x_t)$.

This is limiting when the correct prediction at t depends also on the **future context** $(x_{t+1}, \ldots, x_T)$.

Typical cases (offline sequence labeling): named entity recognition, speech/handwriting 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):

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

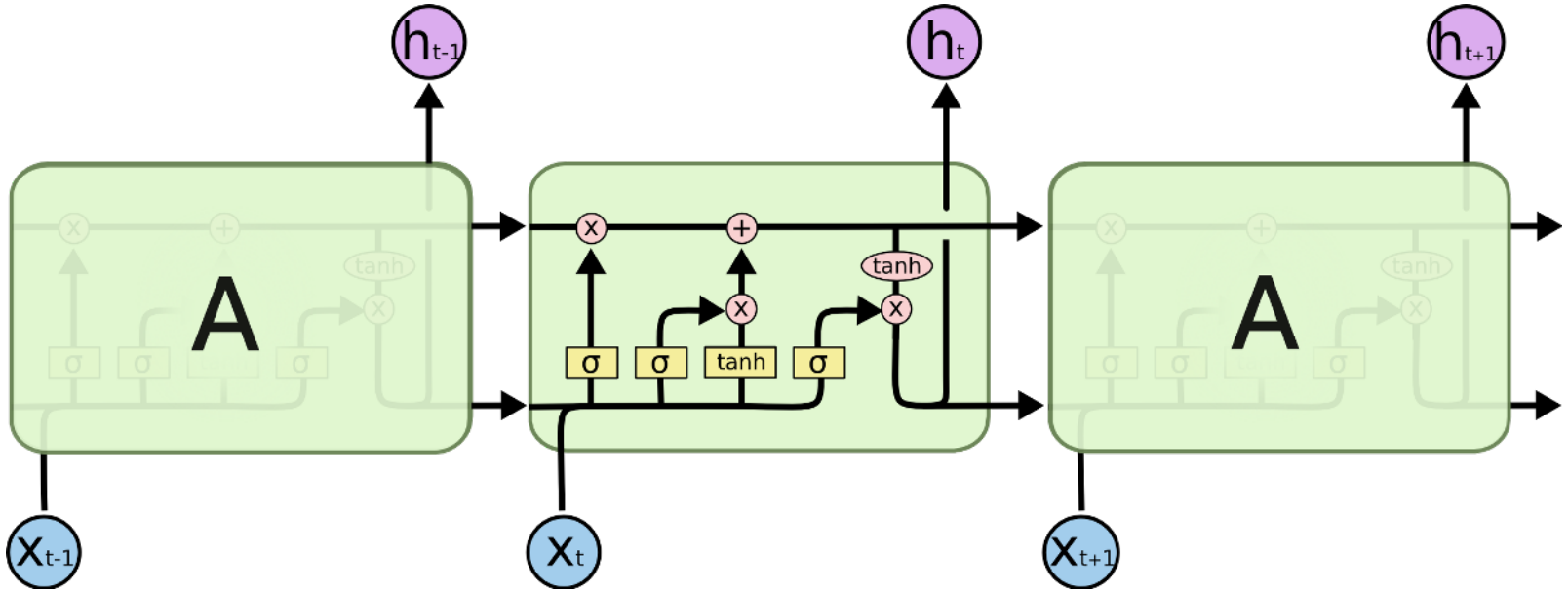
Limitation: requires the full sequence (not suitable for strict online/streaming inference).

Recurrent Neural Networks

Gated RNNs

LSTM

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**.

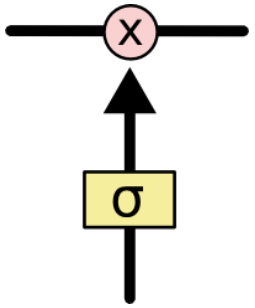


The cell state can remember values **over long time intervals**, and the three gates regulate the flow of information into and out of the cell.

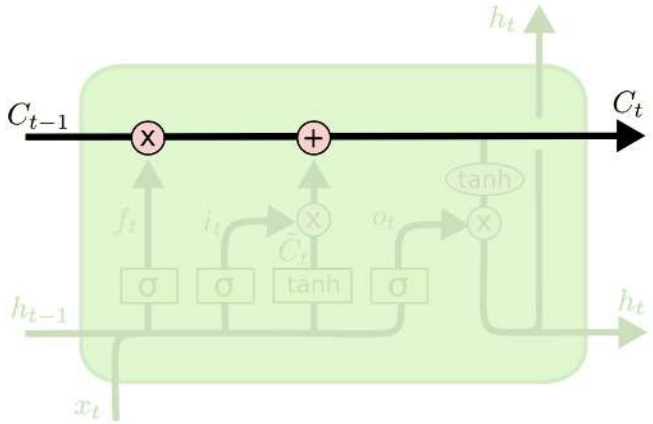
S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," Neural Computation, 1997

Gates

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



$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \odot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} \\ a_{21}b_{21} & a_{22}b_{22} \end{bmatrix}$$

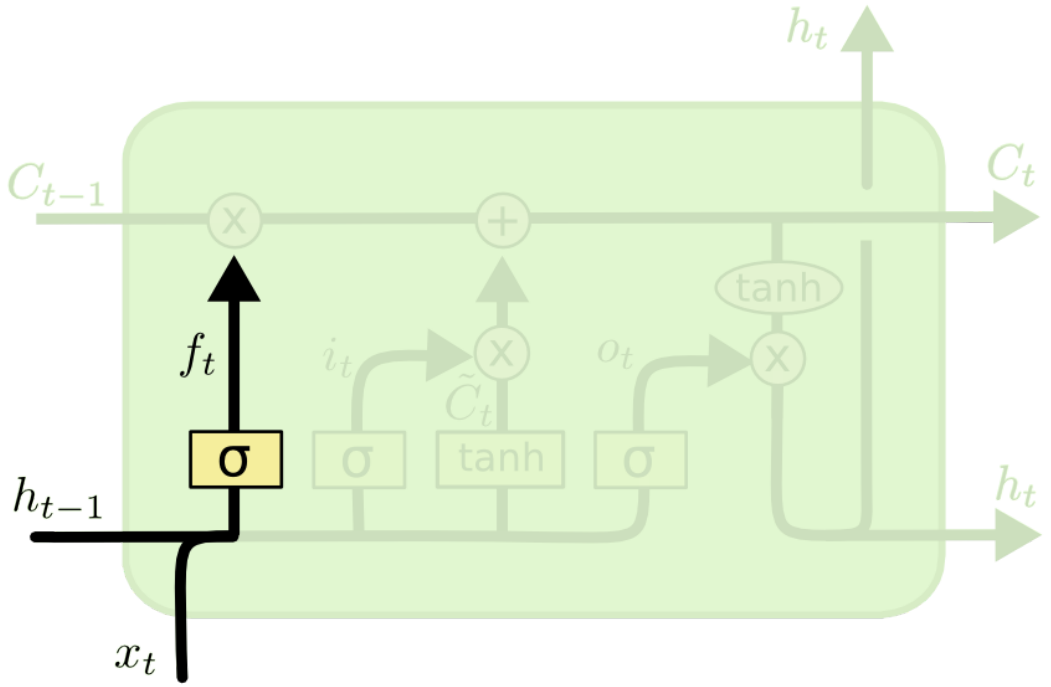


Gates have the ability to remove or add information to the cell state, where the **information can flow with only minor linear interactions.**

Forget Gate

The **forget gate** decides which parts of the long-term state $\mathbf{C}_{t-1}$ should be erased, by looking at $\mathbf{h}_{t-1}$ and $\mathbf{x}_t$.

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



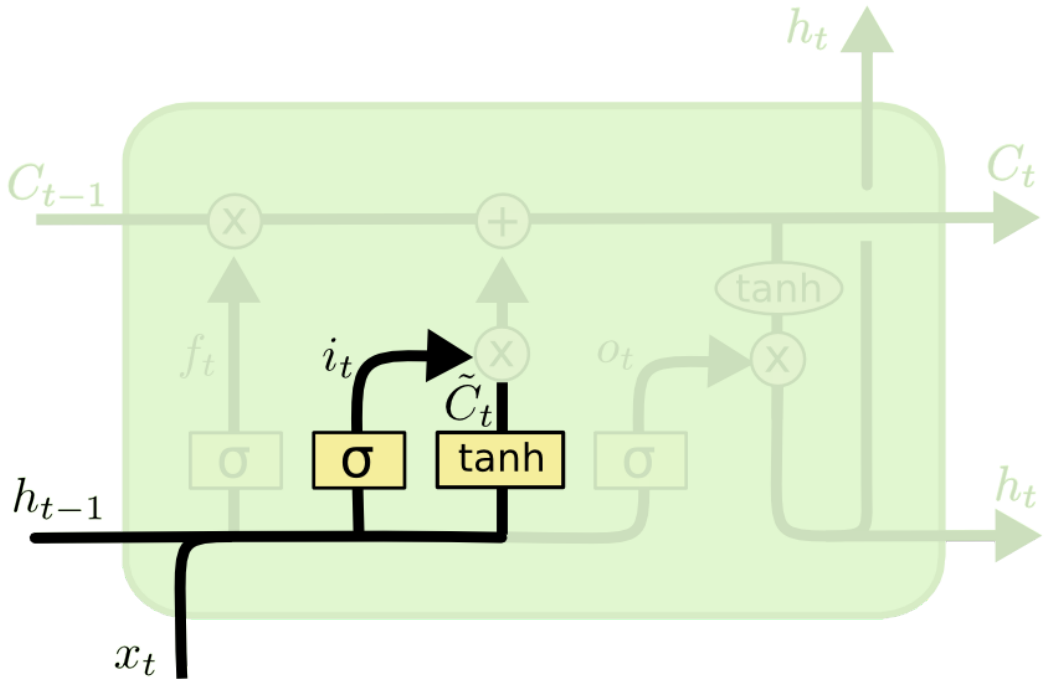
New Memory Generation - Input Gate

To decide what new information to store in the cell state, there are:

- an **input gate** deciding which values to update, and
- a **tanh layer** creating new candidate values $\tilde{\mathbf{C}}_t$.

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

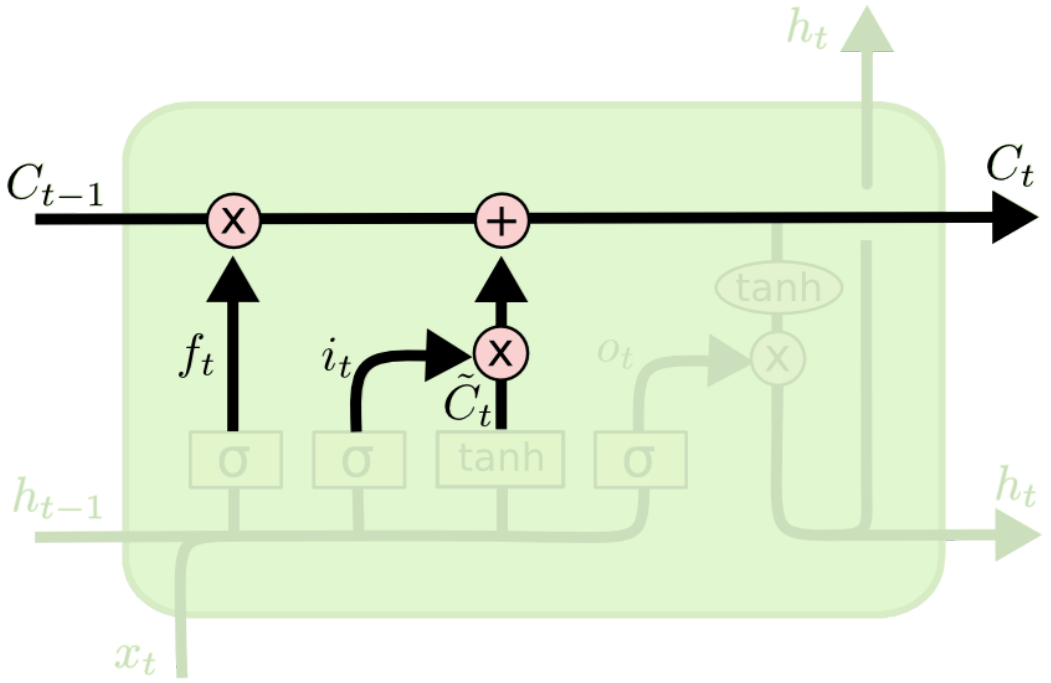
$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{W}_C[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_C)$$



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:

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

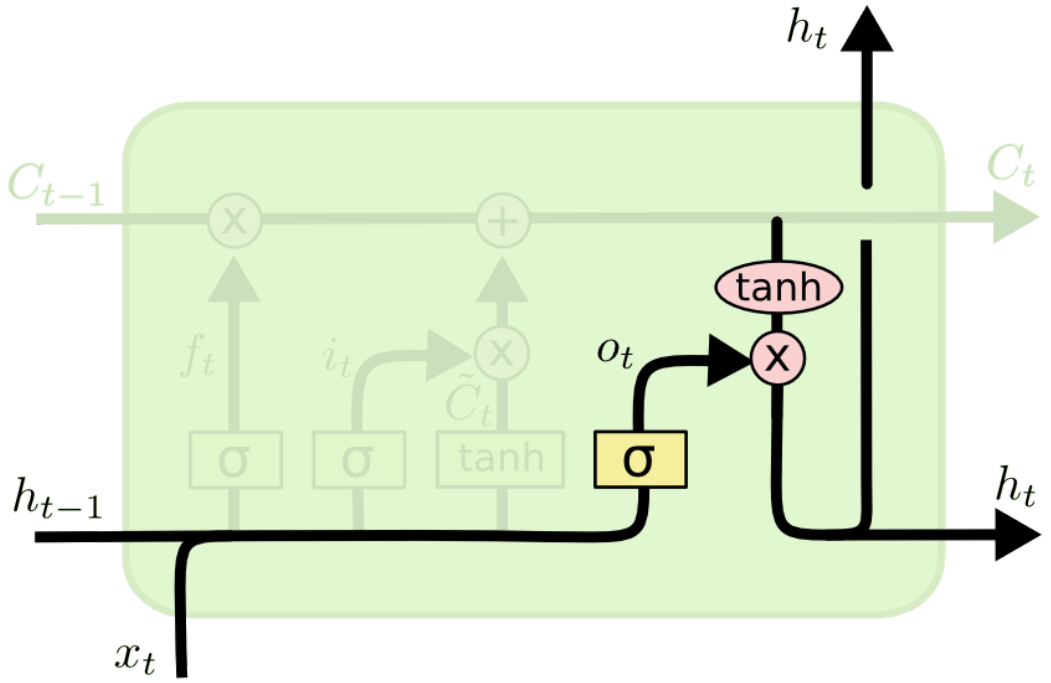


Output Gate

The **output gate** decides which parts of the cell state are output.
The cell state is passed through a pointwise $\tanh$ and gated.

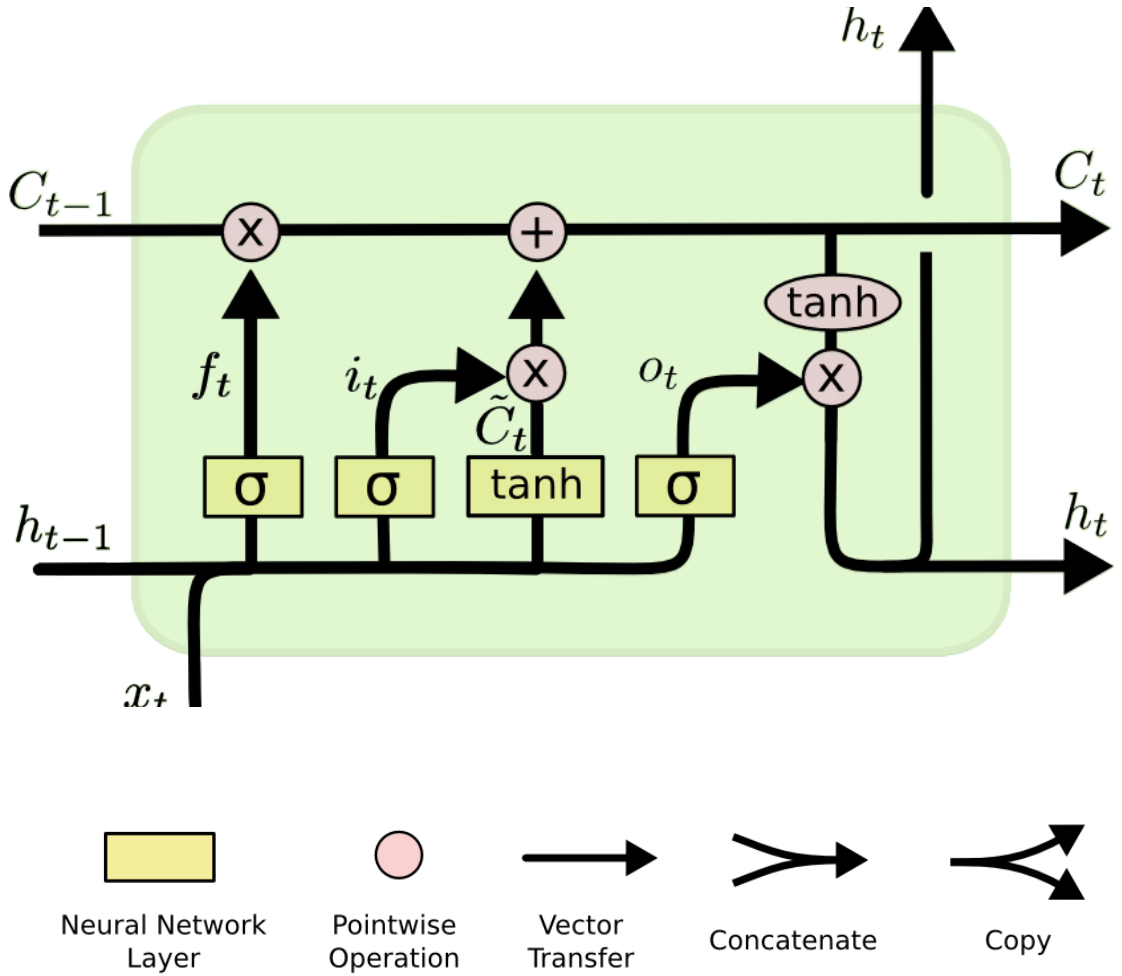
$$\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o)$$

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



LSTM Summary

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



GRU

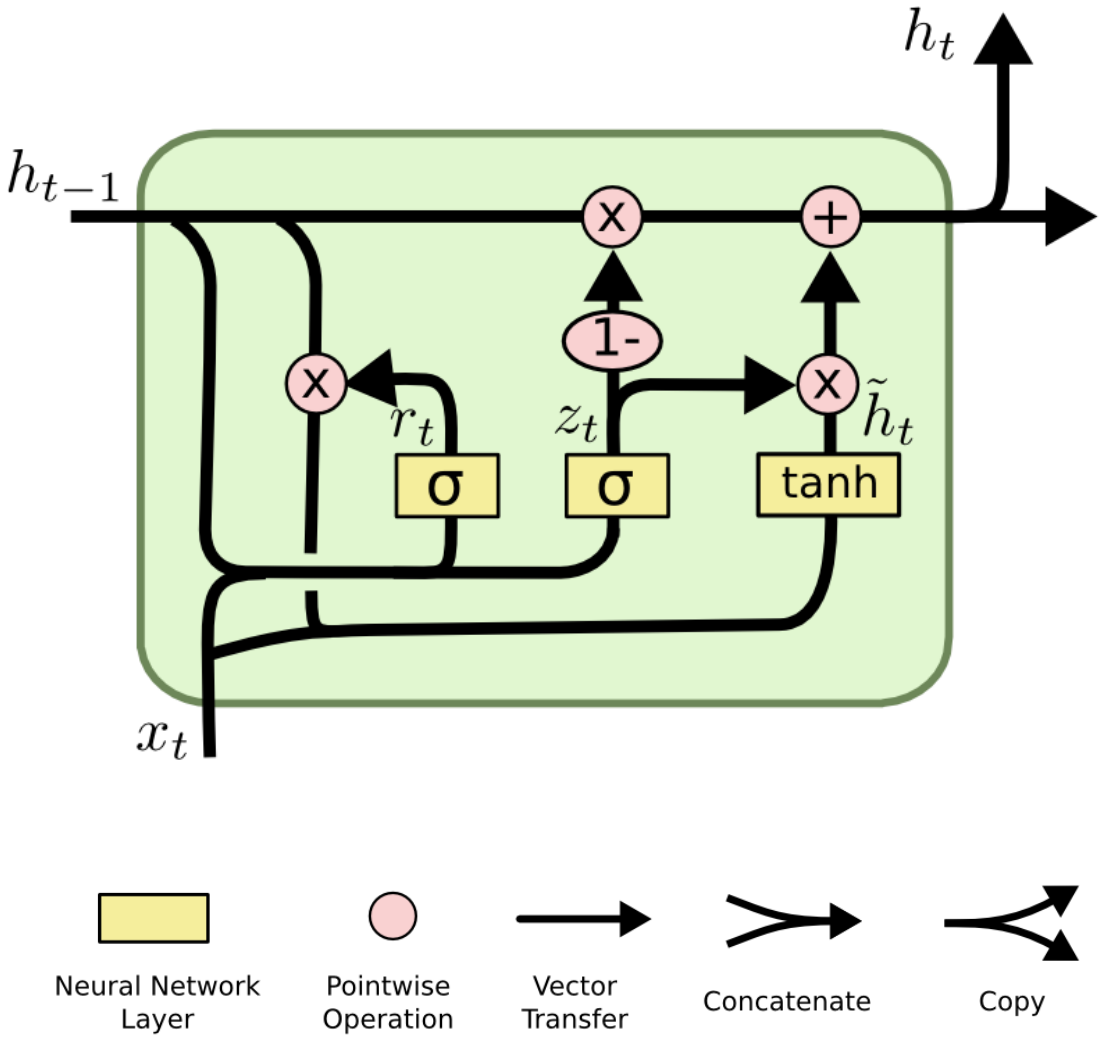
The Gated Recurrent Unit (GRU) uses an **update gate $\mathbf{z}_t$** and a **reset gate $\mathbf{r}_t$**, and merges the cell and hidden states.

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

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

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

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

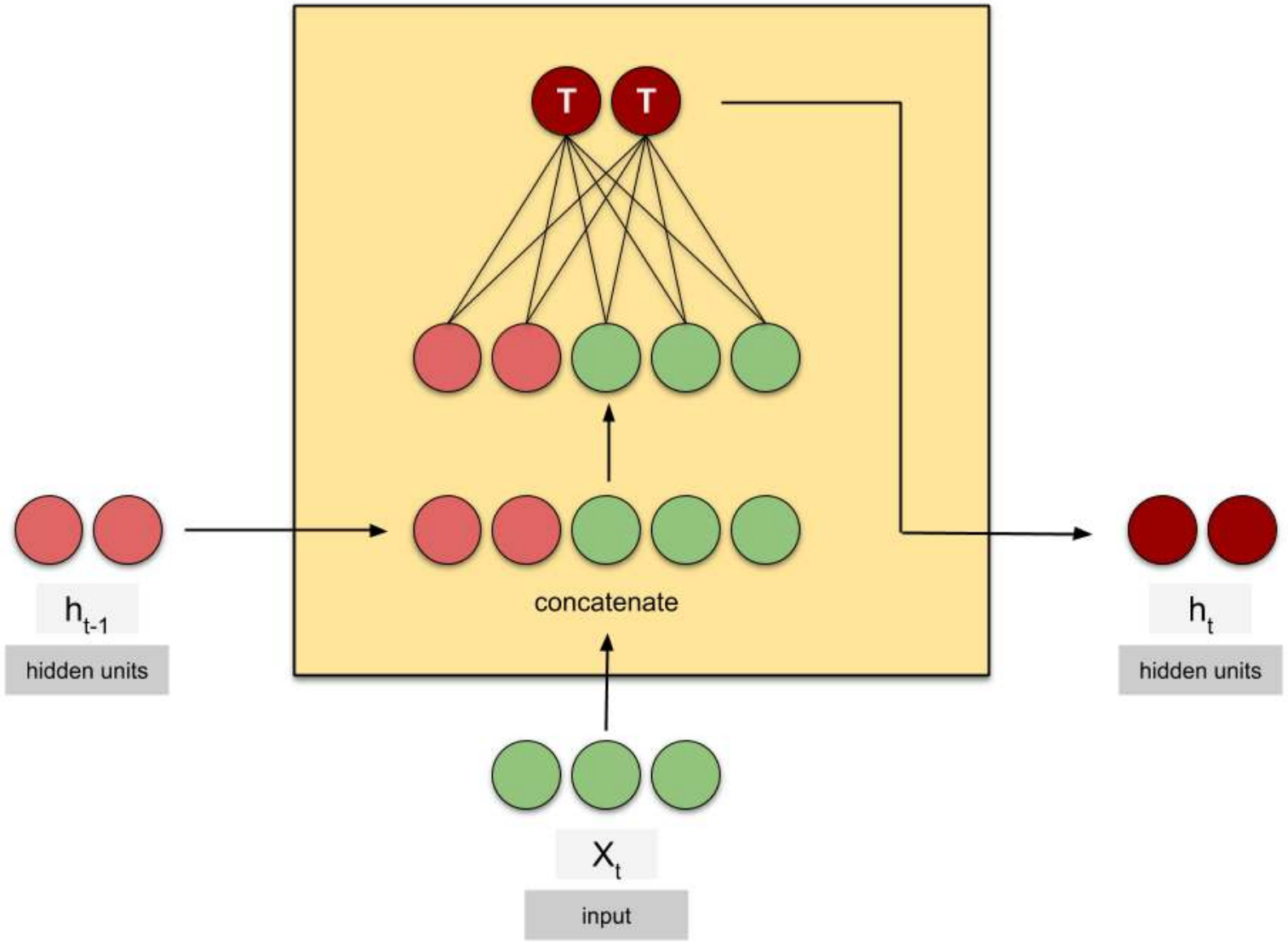


K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014”

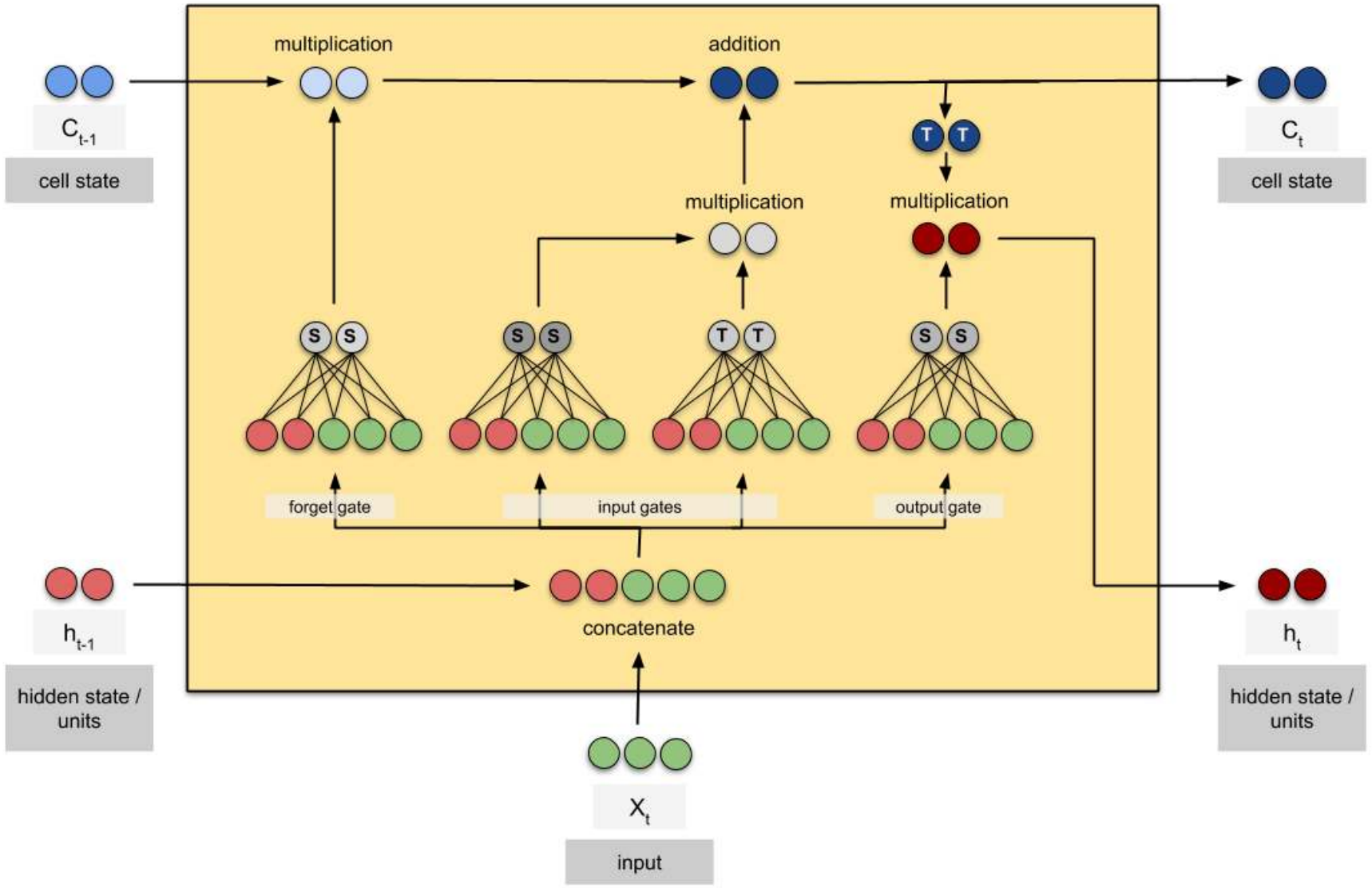
Summary

- sequential data depends on observation order
- standard neural networks are not well suited for sequences
- RNNs pass information across time and share weights
- vanilla RNNs struggle with long dependencies
- LSTMs and GRUs mitigate these issues with gating

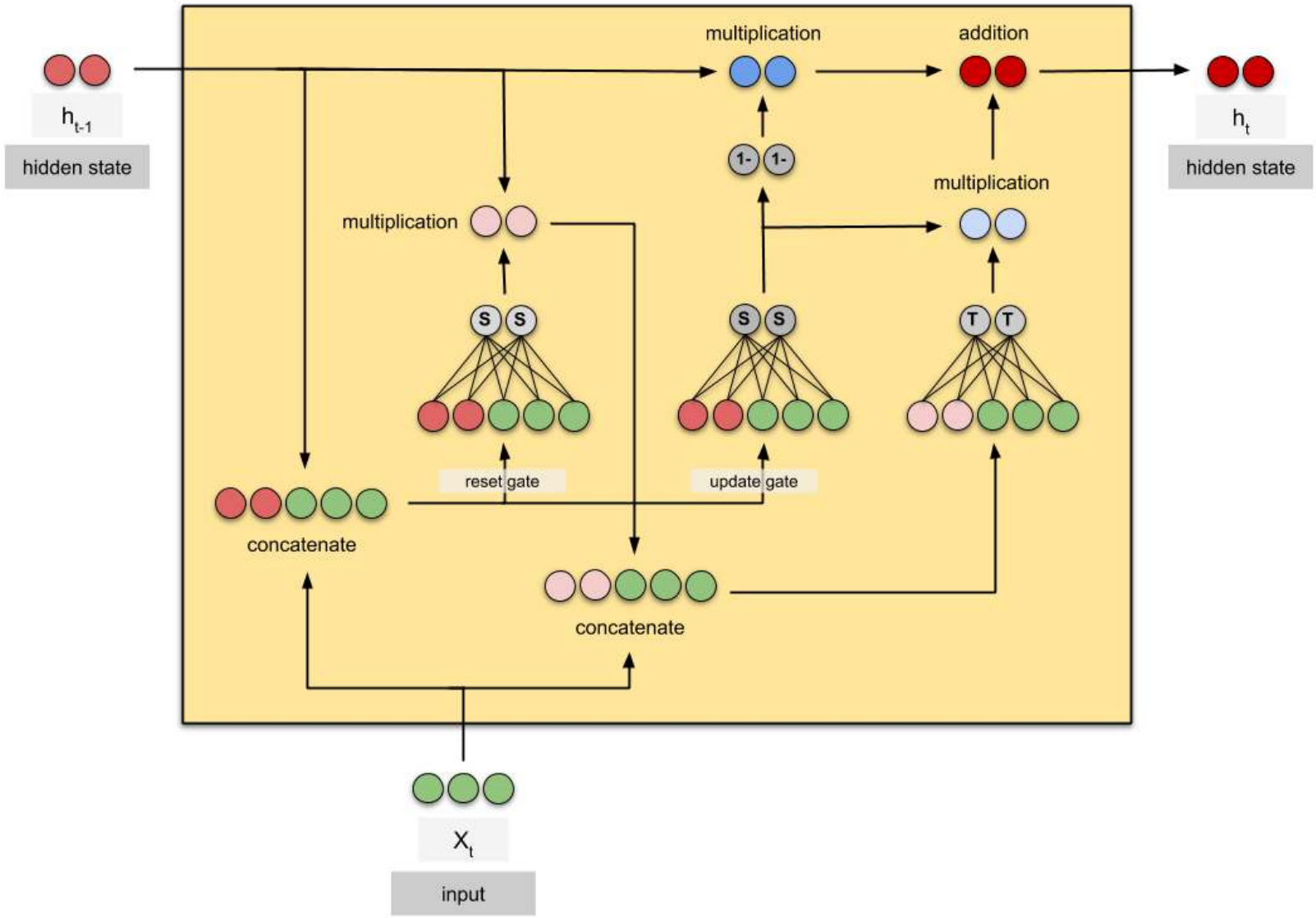
RNN Detail



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
h_t
hidden state
1-
1-
multiplication
multiplication
S
S
S
S
T
T
reset gate
update gate
concatenate
concatenate
X_t
input

Recurrent Neural Networks

Healthcare Applications

In General

RNNs can be useful with sequential data, like time series and text (but as we will see for text they are mostly replaced by transformers).
They can be useful if:

- you have time series with different lengths, or missing values (e.g., recording of patient visits)
- you have short time series (no vanishing gradient problems)
- you want to make and update predictions at each timestamp (many to many network)
- they can be used also as many to one networks (e.g., for classification), but usually other architectures are preferred

Early Detection of Heart Failure

Each raw event is represented as a one-hot vector of 18181 dimensions, which is sequentially fed through a GRU.

The output is a binary class (heart failure / no heart failure)

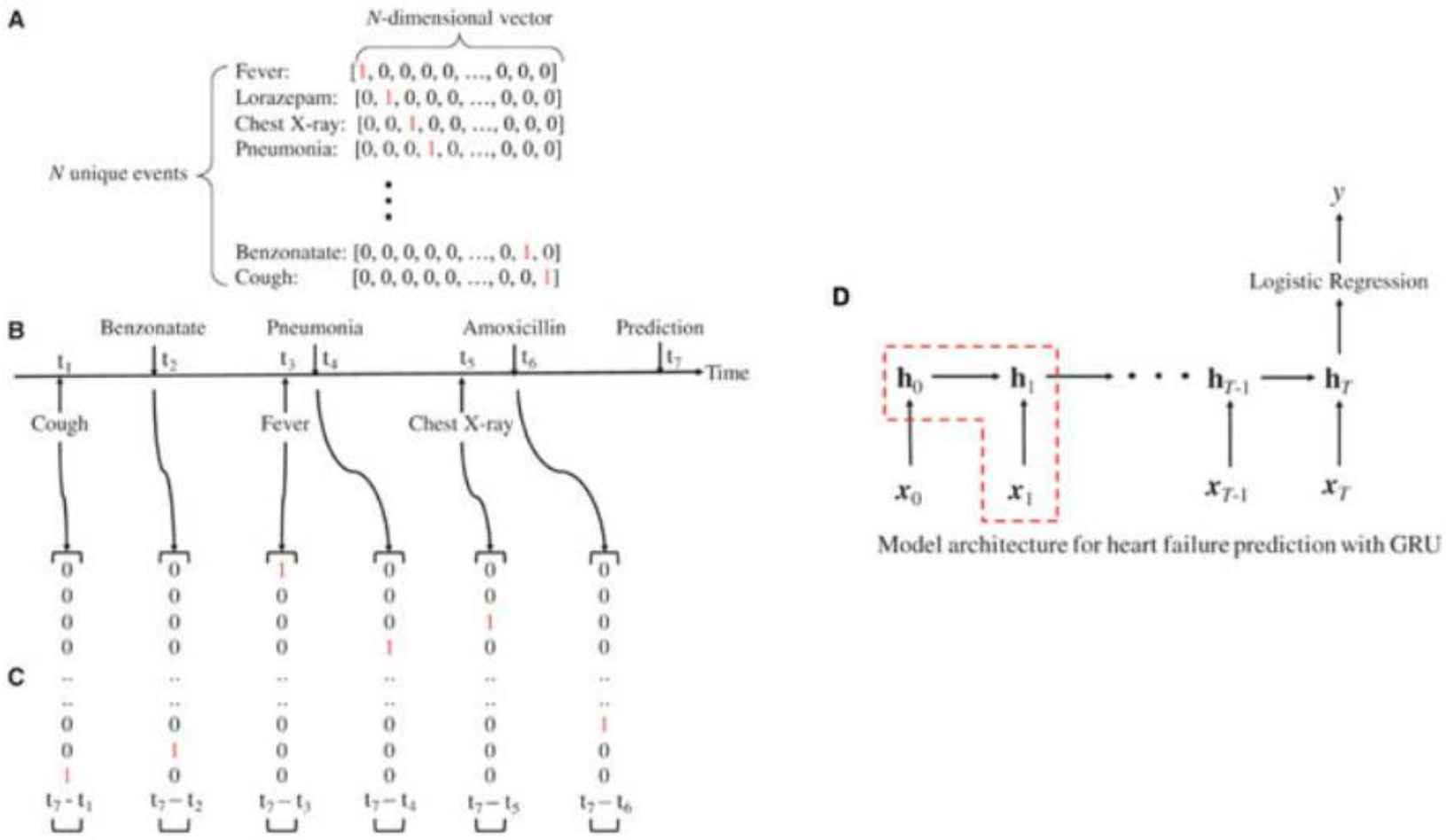
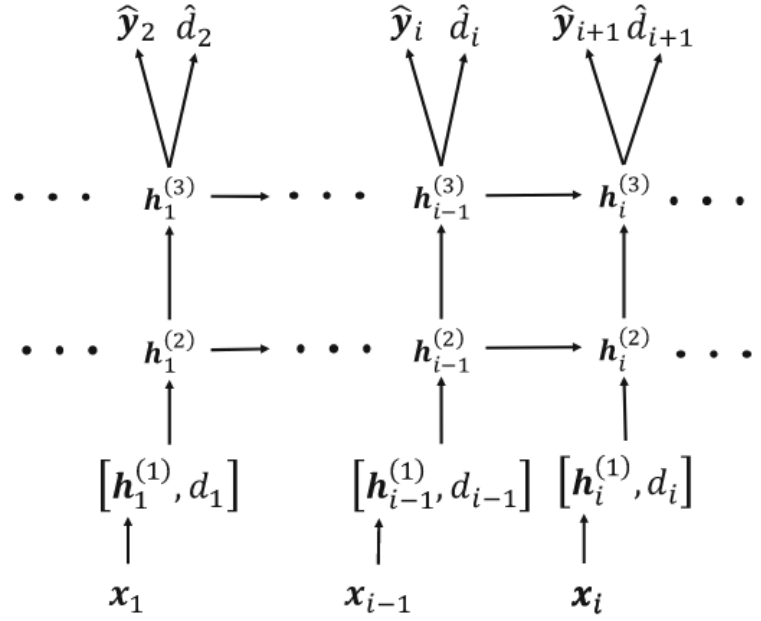


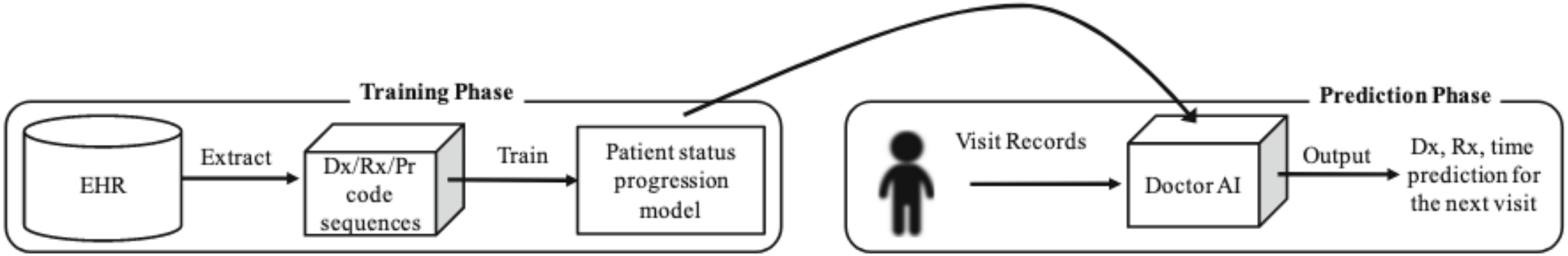
Fig. 7.11 (**a**) One-hot vector encoding of clinical events. (**b**) indicates the time at which the event occurs, assuming we make the prediction at time $t7$. The time duration between current and previous visits are appended as a feature at the end of all input vectors, as shown in (**c**). All the events are passed through a RNN model with binary classification at the end for HF shown in (**d**)

Sequential Clinical Event Prediction

The dataset is EHR data from 265K patients over 8 years from Sutter health, including diagnoses, medications, and procedure information.

- An initial embedding $\mathbf{h}$ is introduced
- Then a 2-layer GRU model is applied to process it.
- The final output is the prediction of diagnosis and medication groups and the duration to the next visit





Choi E, et al. Doctor AI: Predicting Clinical Events via Recurrent Neural Networks. 2016