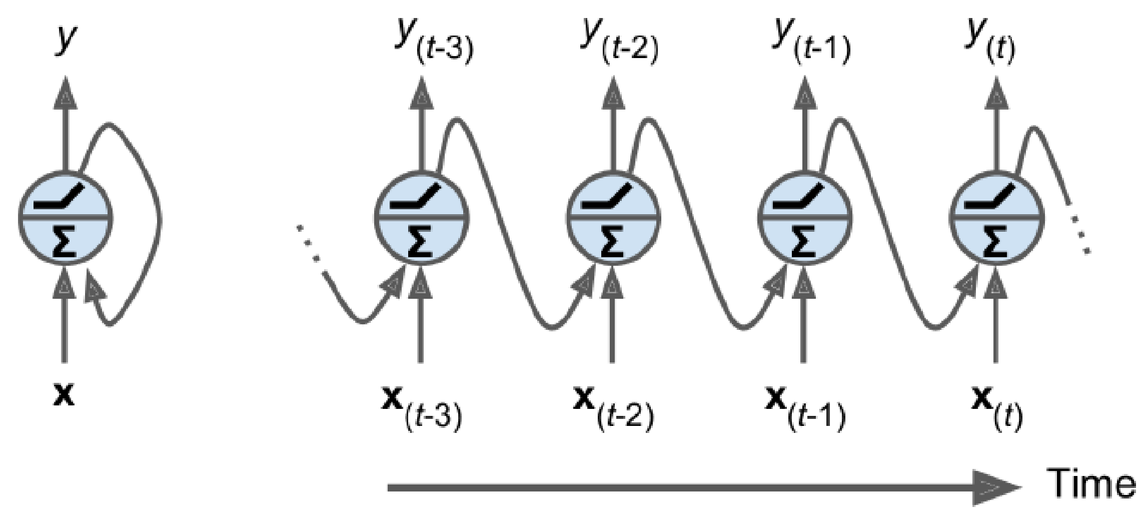
Recurrent Neural Networks (RNN)

Saeed Sharifian

Recurrent Neural Networks

- The idea behind Recurrent neural networks (RNN) is to make use of sequential data.
  - Until here, we assume that all inputs (and outputs) are independent of each other.
  - It is a bad idea for many tasks, e.g., predicting the next word in a sentence (it's better to know which words came before it).
- They can analyze time series data and predict the future.
- They can work on sequences of arbitrary lengths, rather than on fixed-sized inputs.
- Neurons in an RNN have connections pointing backward.
- RNNs have memory, which captures information about what has been calculated so far.



Recurrent Neural Networks

We can process a sequence of vectors **x** by applying a **recurrence formula** at every time step:

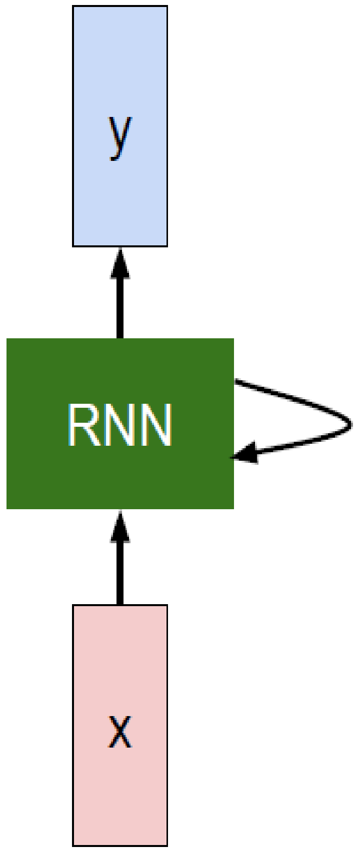
$$\boxed{h_t} = \boxed{f_W}(\boxed{h_{t-1}}, \boxed{x_t})$$

new state

old state

input vector at some time step

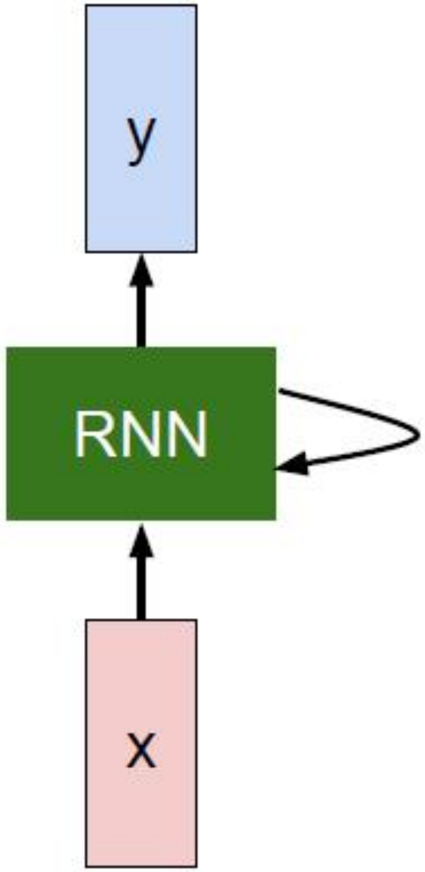
some function with parameters W



Notice: the same function and the same set of parameters are used at every time step.

Recurrent Neural Networks

The state consists of a single *"hidden"* vector **h**:



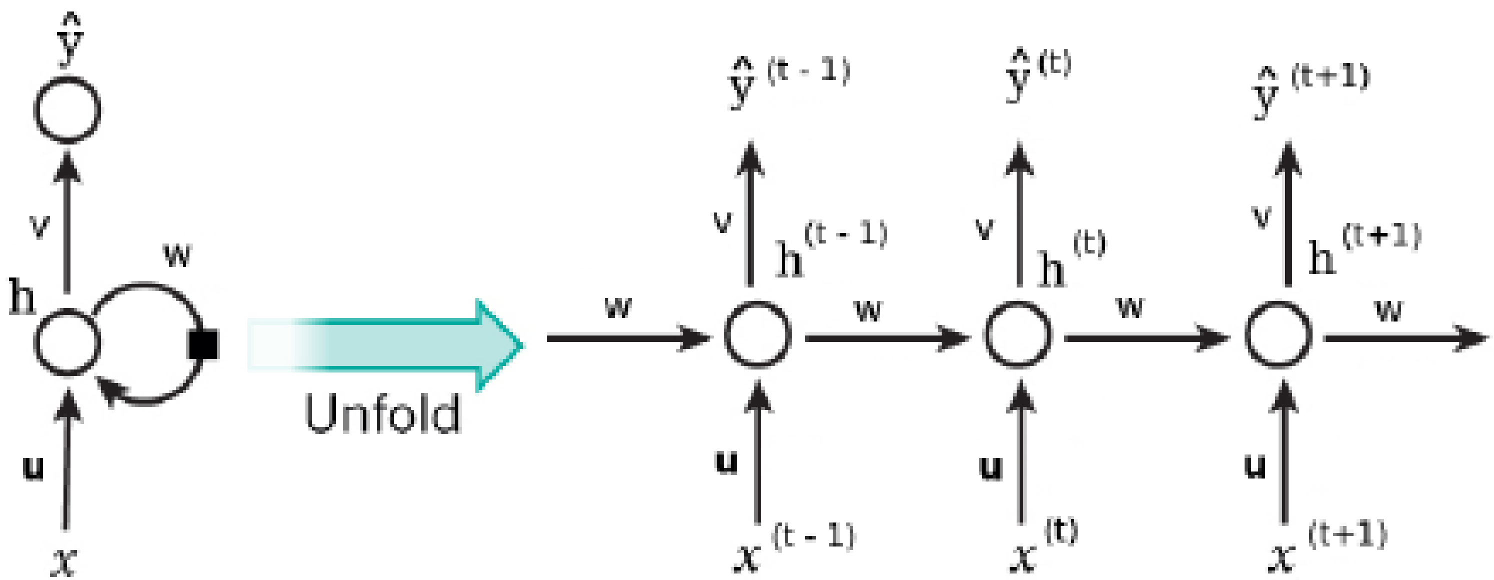
$$h_t = f_W(h_{t-1}, x_t)$$

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

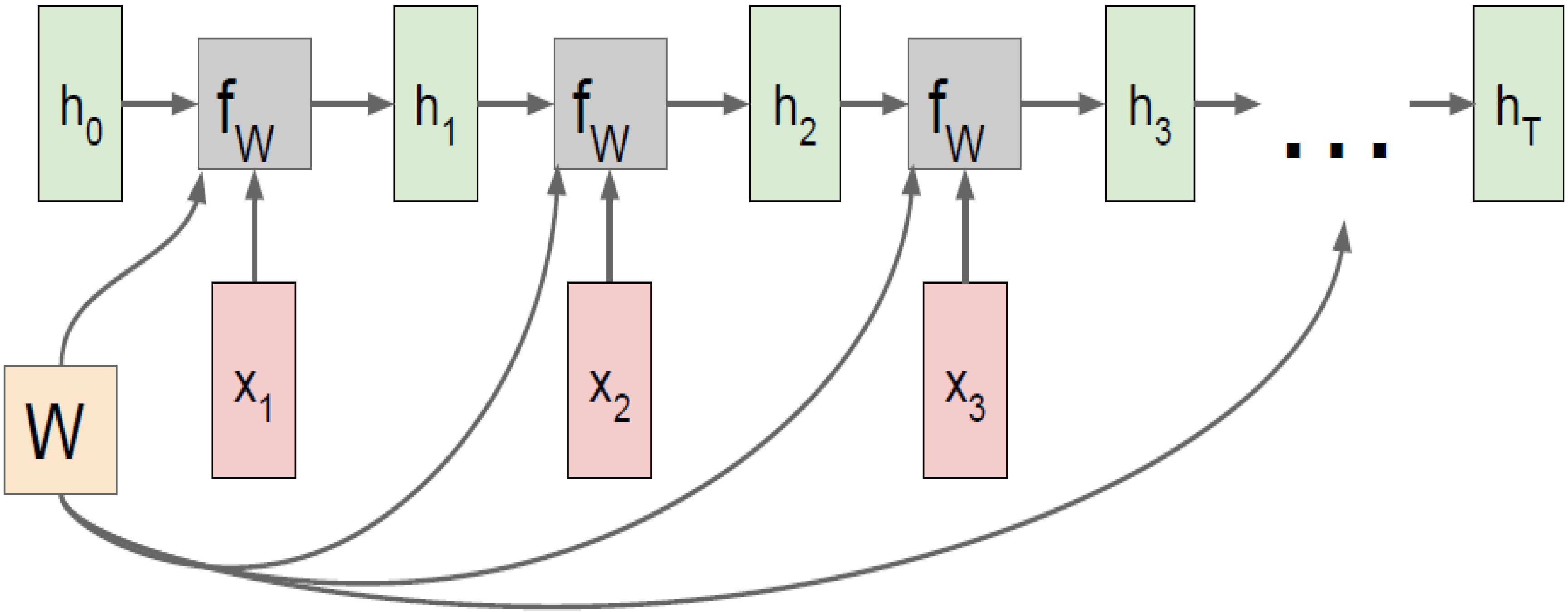
$$y_t = W_{hy}h_t$$

Recurrent Neural Networks

- Unfolding the network: represent a network against the time axis.
  - We write out the network for the complete sequence.
- For example, if the sequence we care about is a sentence of three words, the network would be unfolded into a 3-layer neural network.
  - One layer for each word.



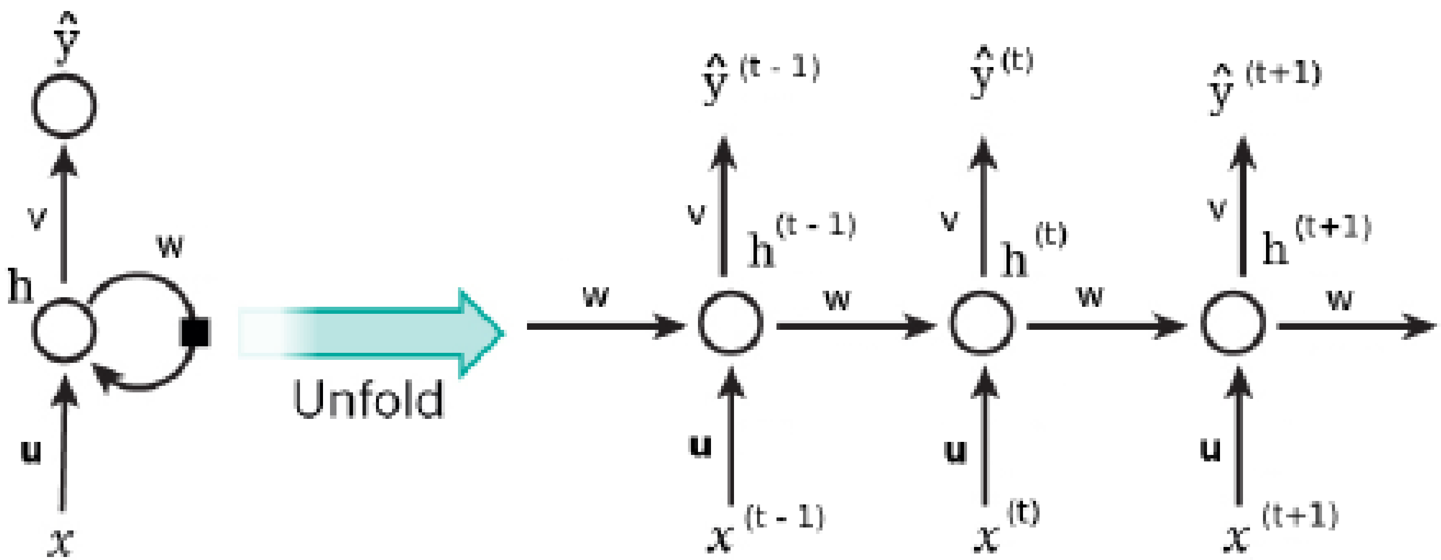
Recurrent Neural Networks



Re-use the same weight matrix at every time-step

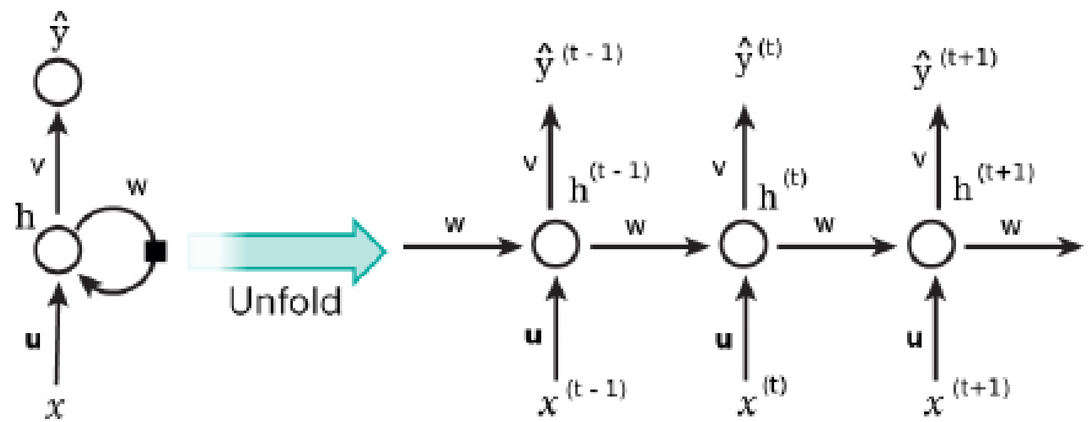
Recurrent Neural Networks

- $h^{(t)} = f(u^{\top}x^{(t)} + wh^{(t-1)})$, where f is an activation function, e.g., tanh or ReLU.
- $\hat{y}^{(t)} = g(vh^{(t)})$, where g can be the softmax function.
- $\text{cost}(y^{(t)}, \hat{y}^{(t)}) = \text{cross\_entropy}(y^{(t)}, \hat{y}^{(t)}) = -\sum y^{(t)} \log \hat{y}^{(t)}$
- $y^{(t)}$ is the correct word at time step t, and $\hat{y}^{(t)}$ is the prediction.



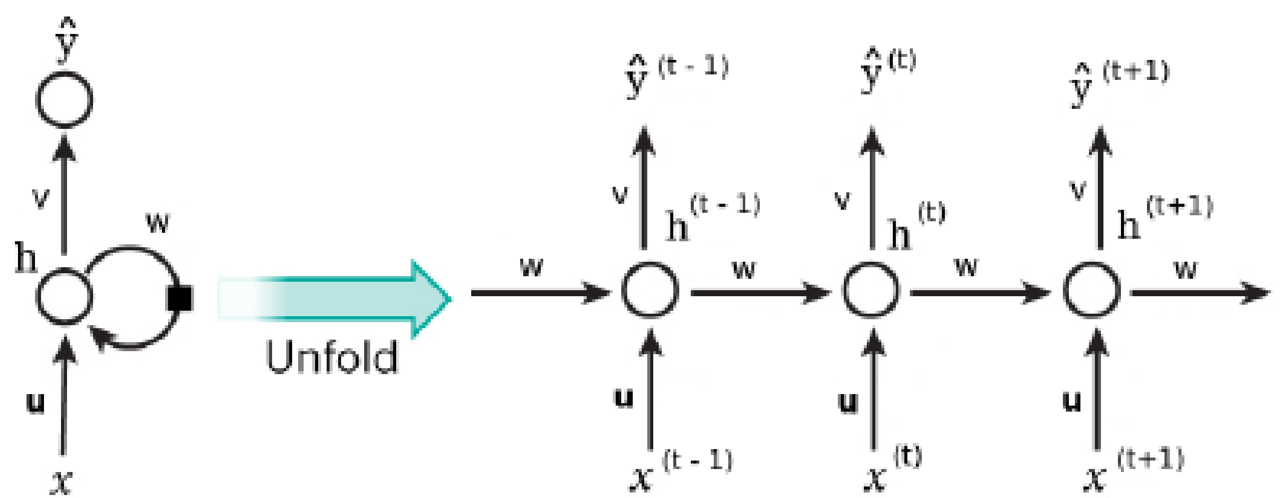
Recurrent Neurons - Weights

- Each recurrent neuron has three sets of weights: $\mathbf{u}$, $\mathtt{w}$, and $\mathtt{v}$.
- $\mathbf{u}$: the weights for the inputs $\mathbf{x}^{(t)}$.
- $\mathbf{x}^{(t)}$: is the input at time step $\mathtt{t}$.
- For example, $\mathbf{x}^{(1)}$ could be a one-hot vector corresponding to the first word of a sentence.
- $\mathtt{w}$: the weights for the hidden state of the previous time step $\mathtt{h}^{(t-1)}$.
- $\mathtt{h}^{(t)}$: is the hidden state (memory) at time step $\mathtt{t}$.
  - $\mathtt{h}^{(t)} = \mathtt{tanh}(\mathbf{u}^{\mathsf{T}}\mathbf{x}^{(t)} + \mathtt{w}\mathtt{h}^{(t-1)})$
  - $\mathtt{h}^{(0)}$ is the initial hidden state.



Recurrent Neurons - Weights

- $\mathtt{v}$: the weights for the hidden state of the current time step $\mathtt{h}^{(t)}$.
- $\hat{\mathtt{y}}^{(t)}$ is the output at step $\mathtt{t}$.
- $\hat{\mathtt{y}}^{(t)} = \mathtt{softmax}(\mathtt{vh}^{(t)})$
- For example, if we wanted to predict the next word in a sentence, it would be a vector of probabilities across our vocabulary.

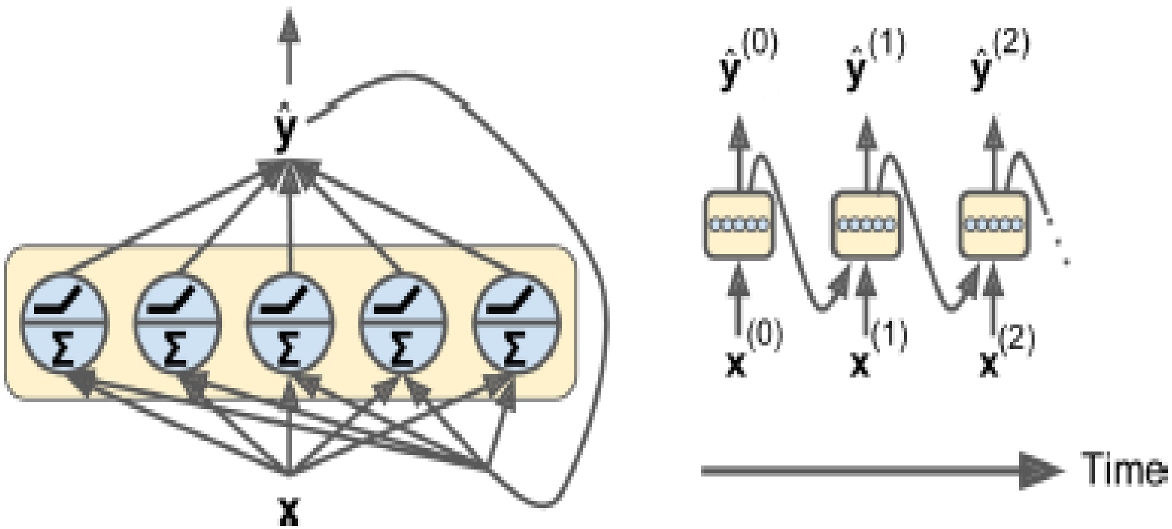


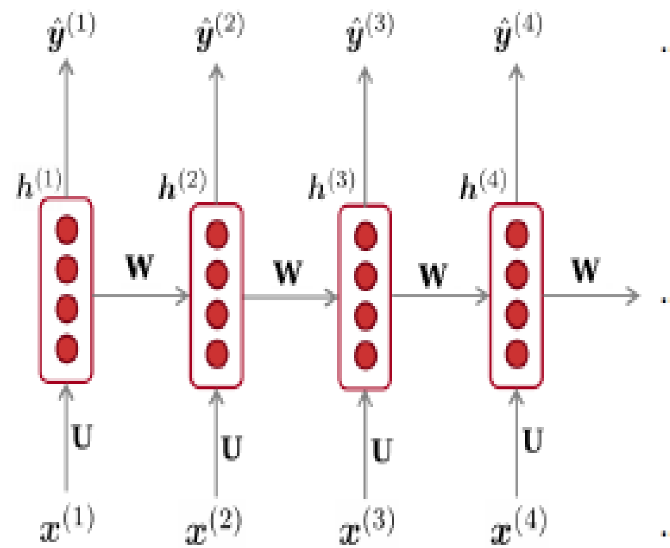
Layers of Recurrent Neurons

- At each time step t, every neuron of a layer receives both the input vector $\mathbf{x}^{(t)}$ and the output vector from the previous time step $\mathbf{h}^{(t-1)}$.

$$\mathbf{h}^{(t)} = \mathtt{tanh}(\mathbf{u}^\intercal \mathbf{x}^{(t)} + \mathbf{w}^\intercal \mathbf{h}^{(t-1)})$$

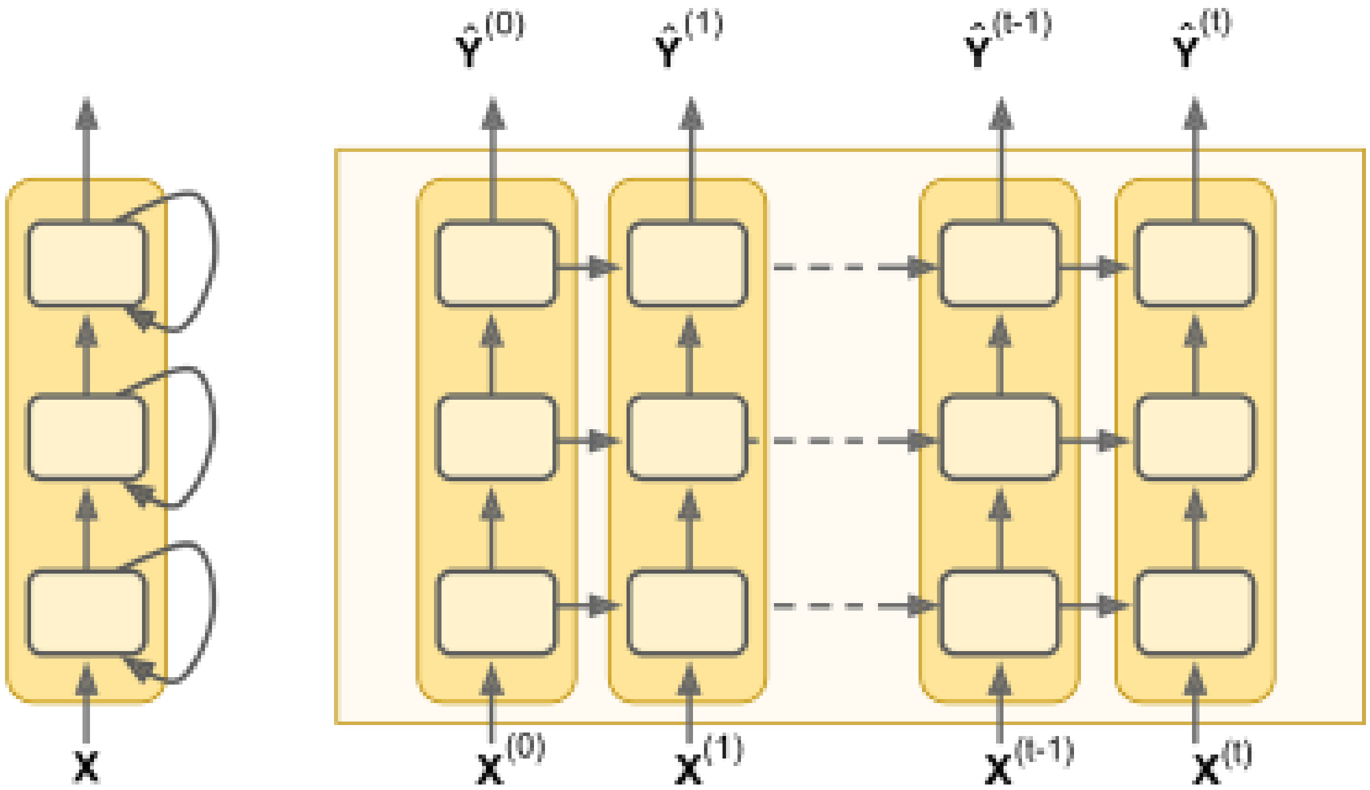
$$\mathbf{y}^{(t)} = \mathtt{sigmoid}(\mathbf{v}^\intercal \mathbf{h}^{(t)})$$





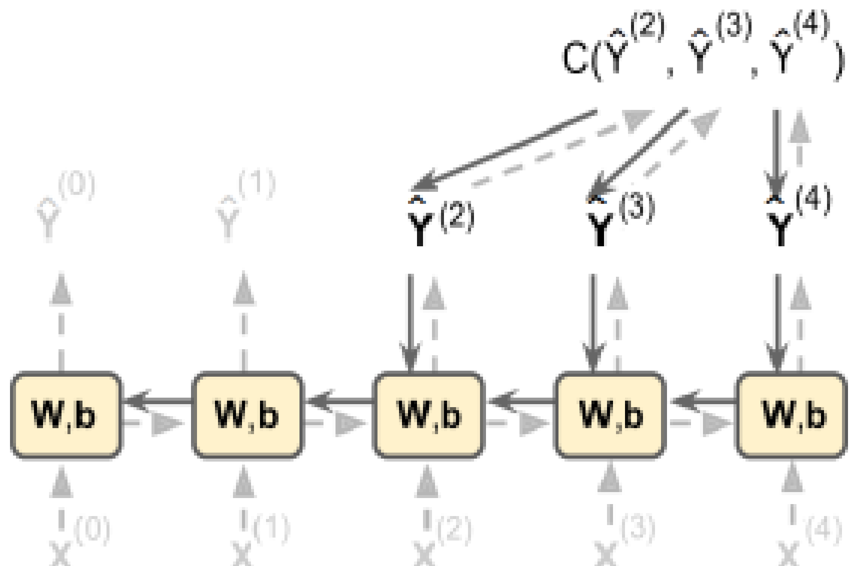
Deep RNN

- Stacking multiple layers of cells gives you a deep RNN.



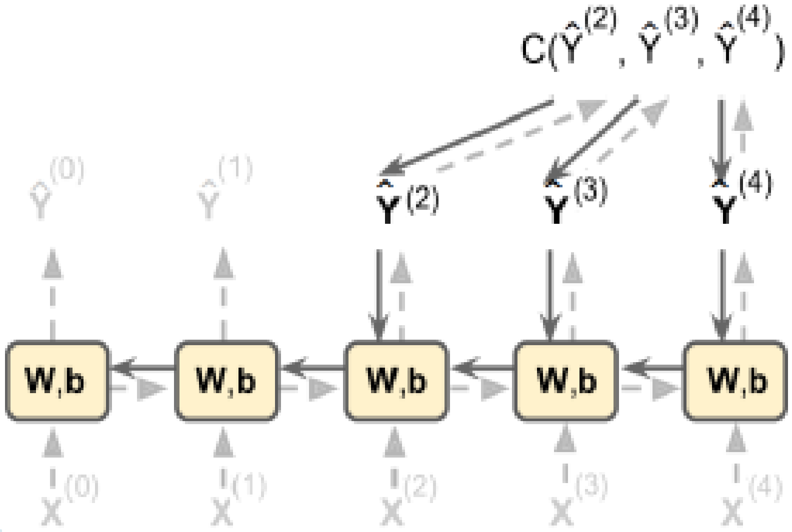
Training RNNs

- To train an RNN, we should unroll it through time and then simply use regular backpropagation.
- This strategy is called backpropagation through time (BPTT).
- To train the model using BPTT, we go through the following steps:
- 1. Forward pass through the unrolled network (represented by the dashed arrows).
- 2. The cost function is $\mathtt{C}(\hat{\mathbf{y}}^{\mathtt{tmin}}, \hat{\mathbf{y}}^{\mathtt{tmin+1}}, \cdots, \hat{\mathbf{y}}^{\mathtt{tmax}})$, where tmin and tmax are the first and last output time steps, not counting the ignored outputs.



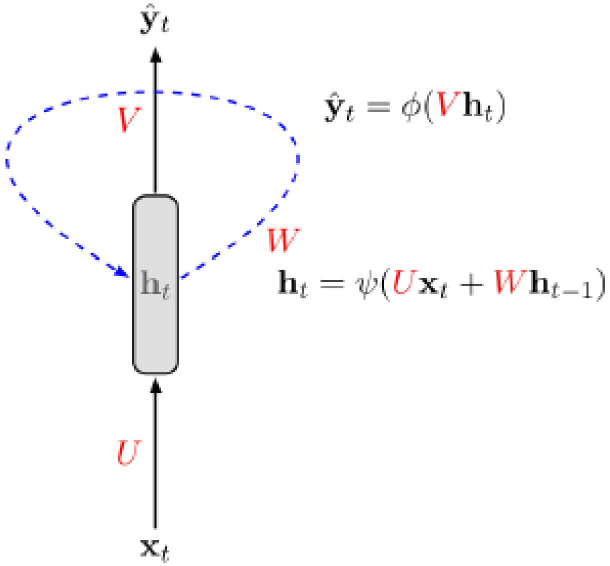
Backpropagation Through Time

- 3. Propagate backward the gradients of that cost function through the unrolled network (represented by the solid arrows).
- 4. The model parameters are updated using the gradients computed during BPTT.
- The gradients flow backward through all the outputs used by the cost function, not just through the final output.
- For example, in the following figure:
  - The cost function is computed using the last three outputs, $\hat{\mathbf{y}}^{(2)}$, $\hat{\mathbf{y}}^{(3)}$, and $\hat{\mathbf{y}}^{(4)}$.
  - Gradients flow through these three outputs, but not through $\hat{\mathbf{y}}^{(0)}$ and $\hat{\mathbf{y}}^{(1)}$.

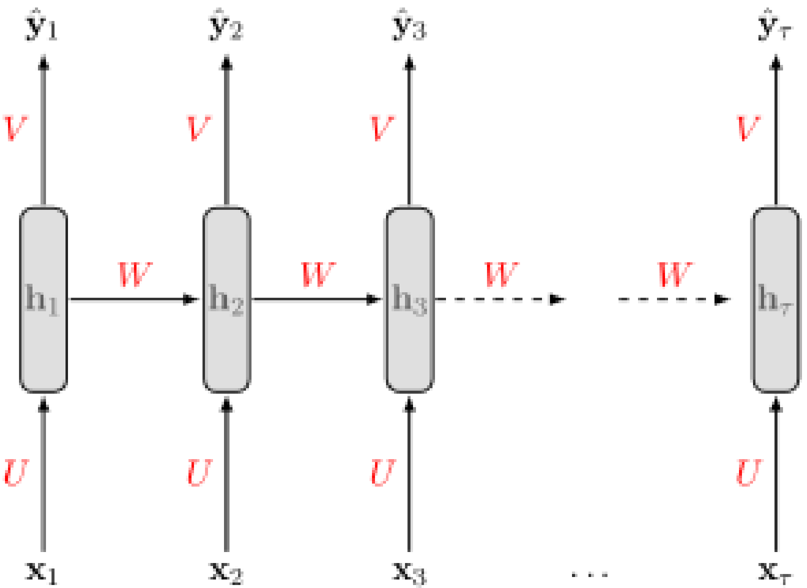


BPTT Step by Step

$$s^{(t)} = u^{T}x^{(t)} + wh^{(t-1)}$$
$$h^{(t)} = \tanh(s^{(t)})$$
$$z^{(t)} = vh^{(t)}$$
$$\hat{y}^{(t)} = \mathrm{softmax}(z^{(t)})$$
$$J^{(t)} = \mathrm{cross\_entropy}(y^{(t)}, \hat{y}^{(t)}) = -\sum y^{(t)} \log \hat{y}^{(t)}$$



- We treat the full sequence as one training example.
- The total error E is just the sum of the errors at each time step.
- E.g., $E = J^{(1)} + J^{(2)} + \cdots + J^{(t)}$



RNN Design Patterns

Vector-to-Sequence

- Vector-to-sequence network: takes a single input at the first time step, and let it output a sequence.
- E.g., the input could be an image, and the output could be a caption for that image.

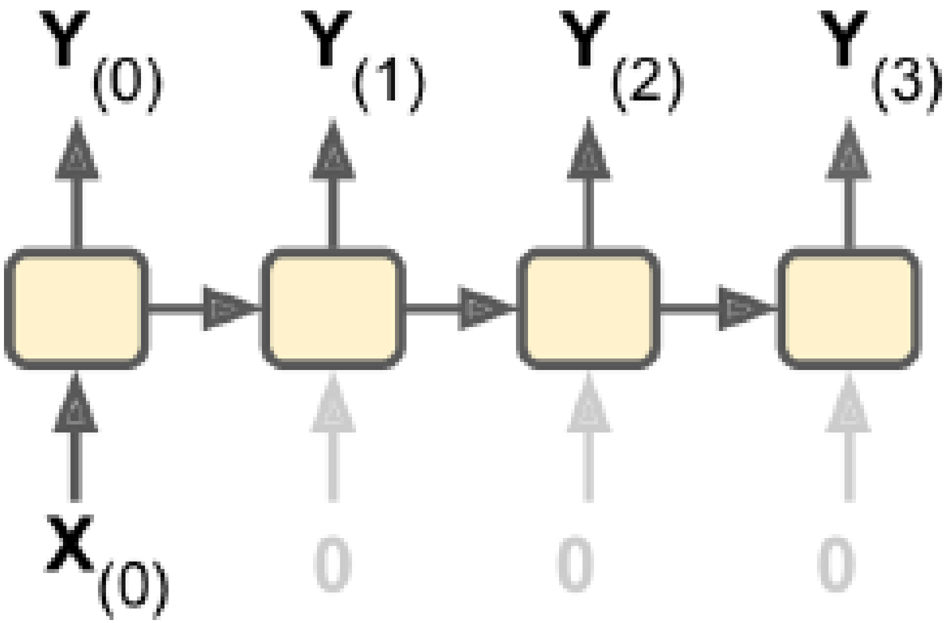
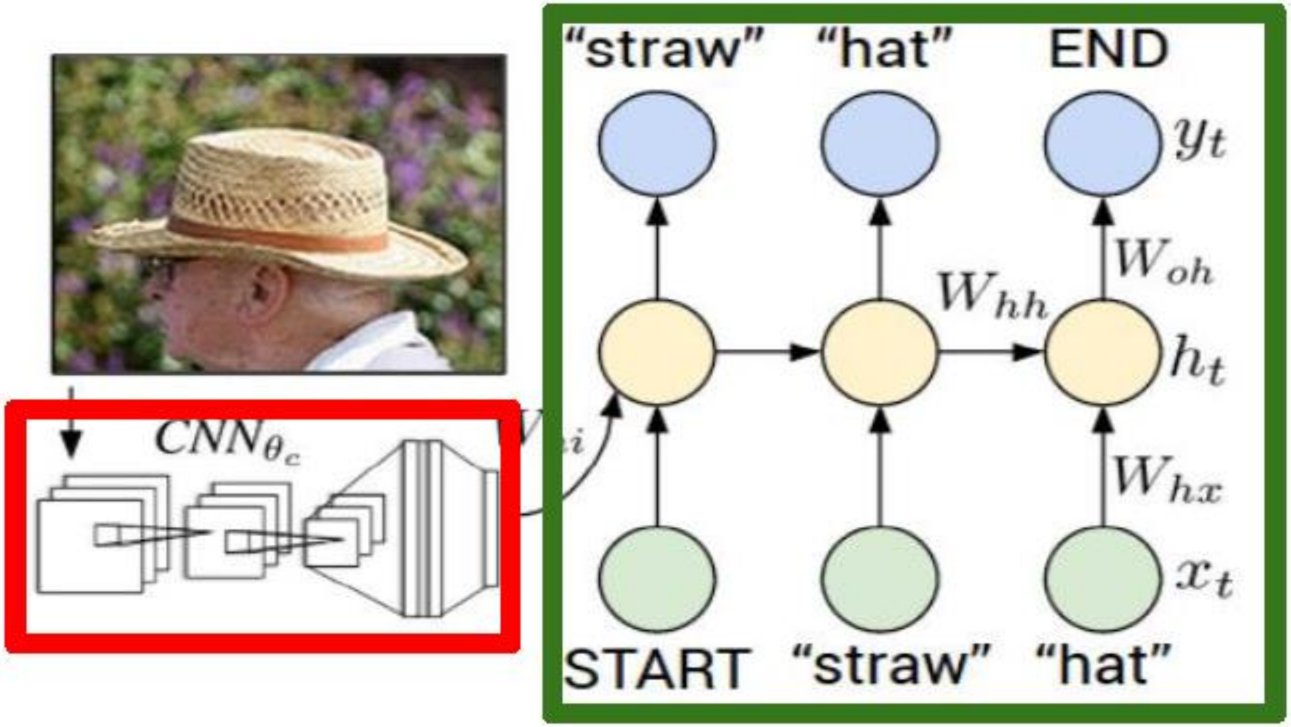


Image Captioning

image -> sequence of words

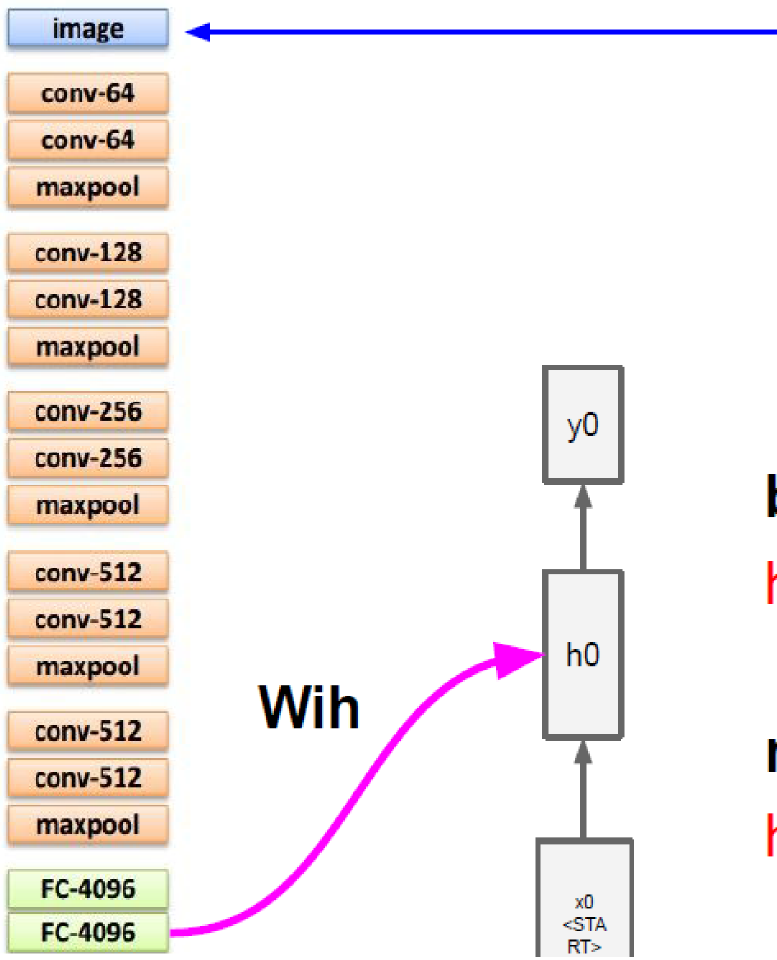
Image Captioning

Recurrent Neural Network



Convolutional Neural Network

Image Captioning



test image

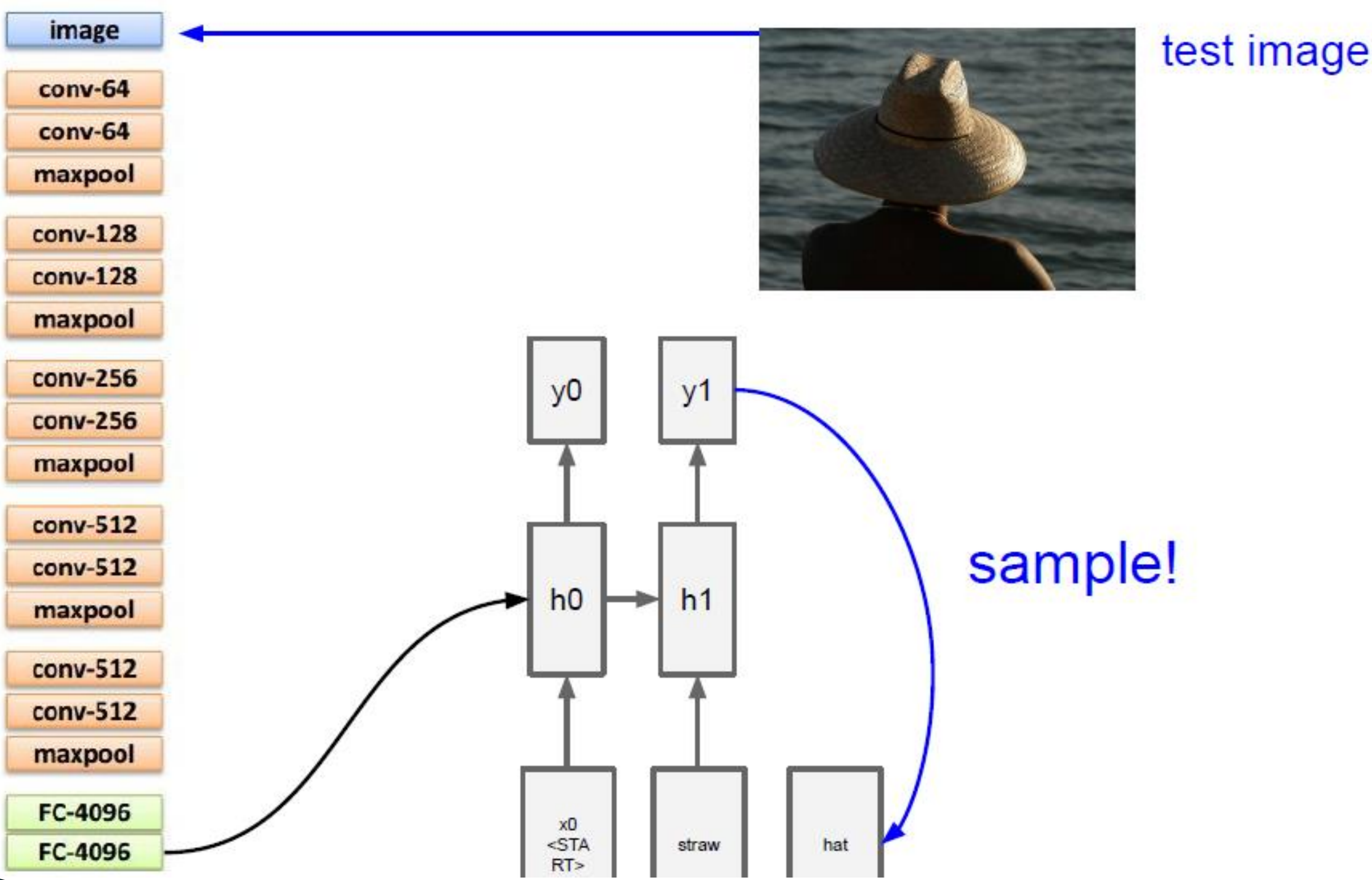
before:

h = tanh(Wxh * x + Whh * h)

now:

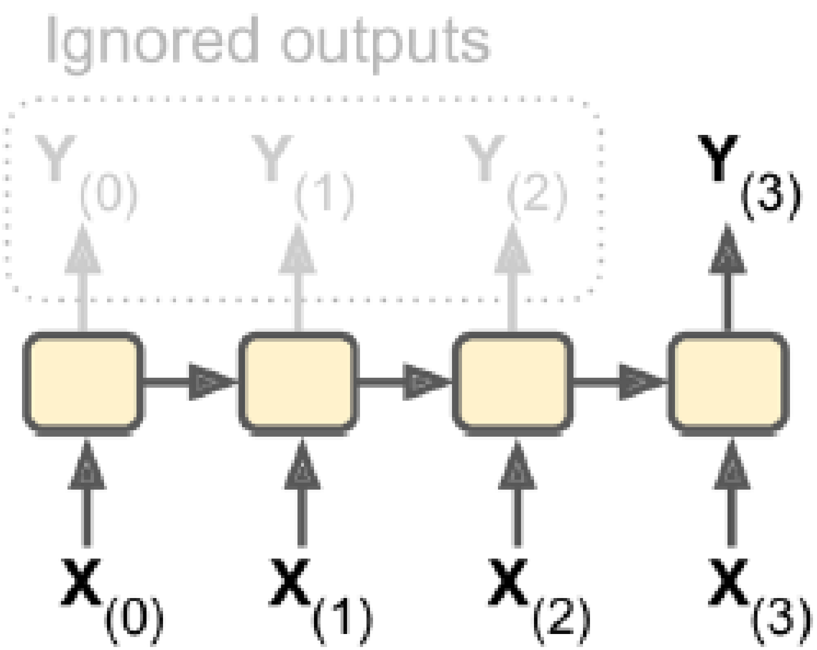
h = tanh(Wxh * x + Whh * h + **Wih** * **v**)

Image Captioning



Sequence-to-Vector

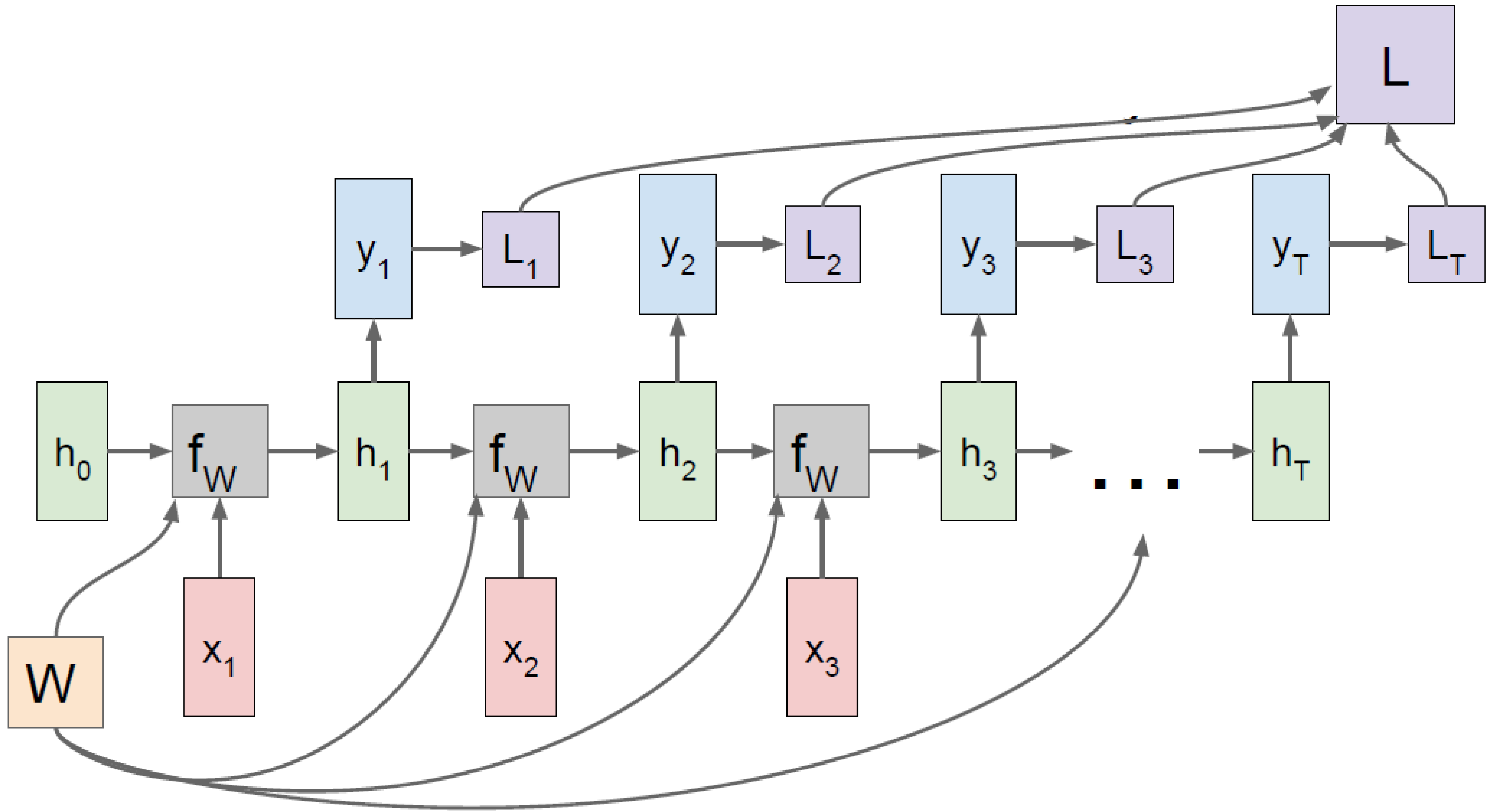
- Sequence-to-vector network: takes a sequence of inputs, and ignore all outputs except for the last one.
- E.g., you could feed the network a sequence of words corresponding to a movie review, and the network would output a sentiment score.



Sentiment Classification

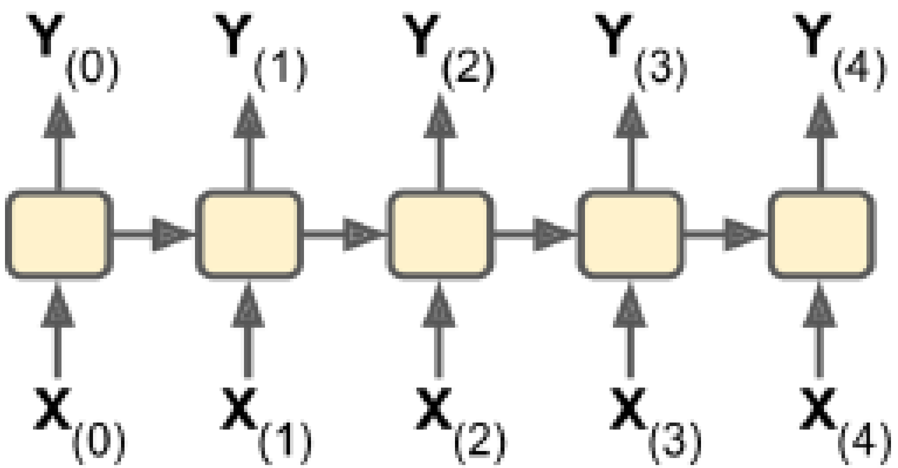
sequence of words -> sentiment

Many to Many



Sequence-to-Sequence

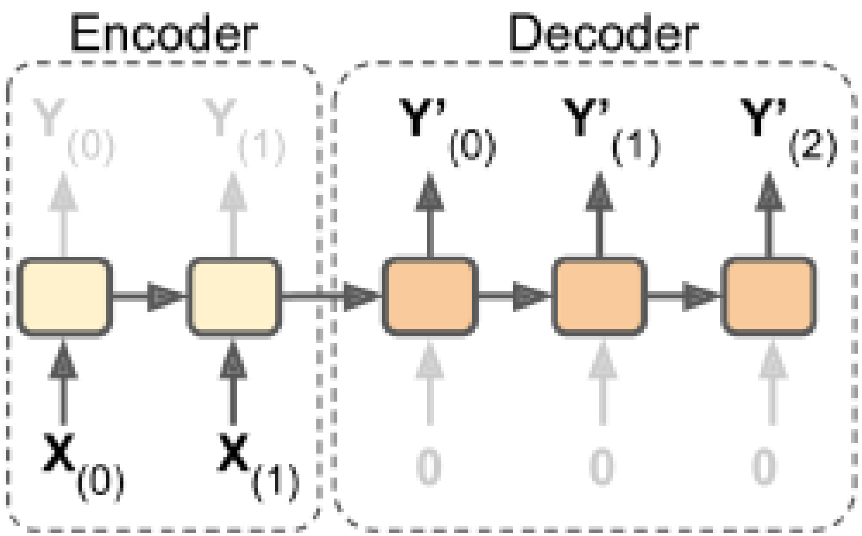
- Sequence-to-sequence network: takes a sequence of inputs and produce a sequence of outputs.
- Useful for predicting time series such as stock prices: you feed it the prices over the last N days, and it must output the prices shifted by one day into the future.
- Here, both input sequences and output sequences have the same length.



Video classification on frame level

Encoder-Decoder

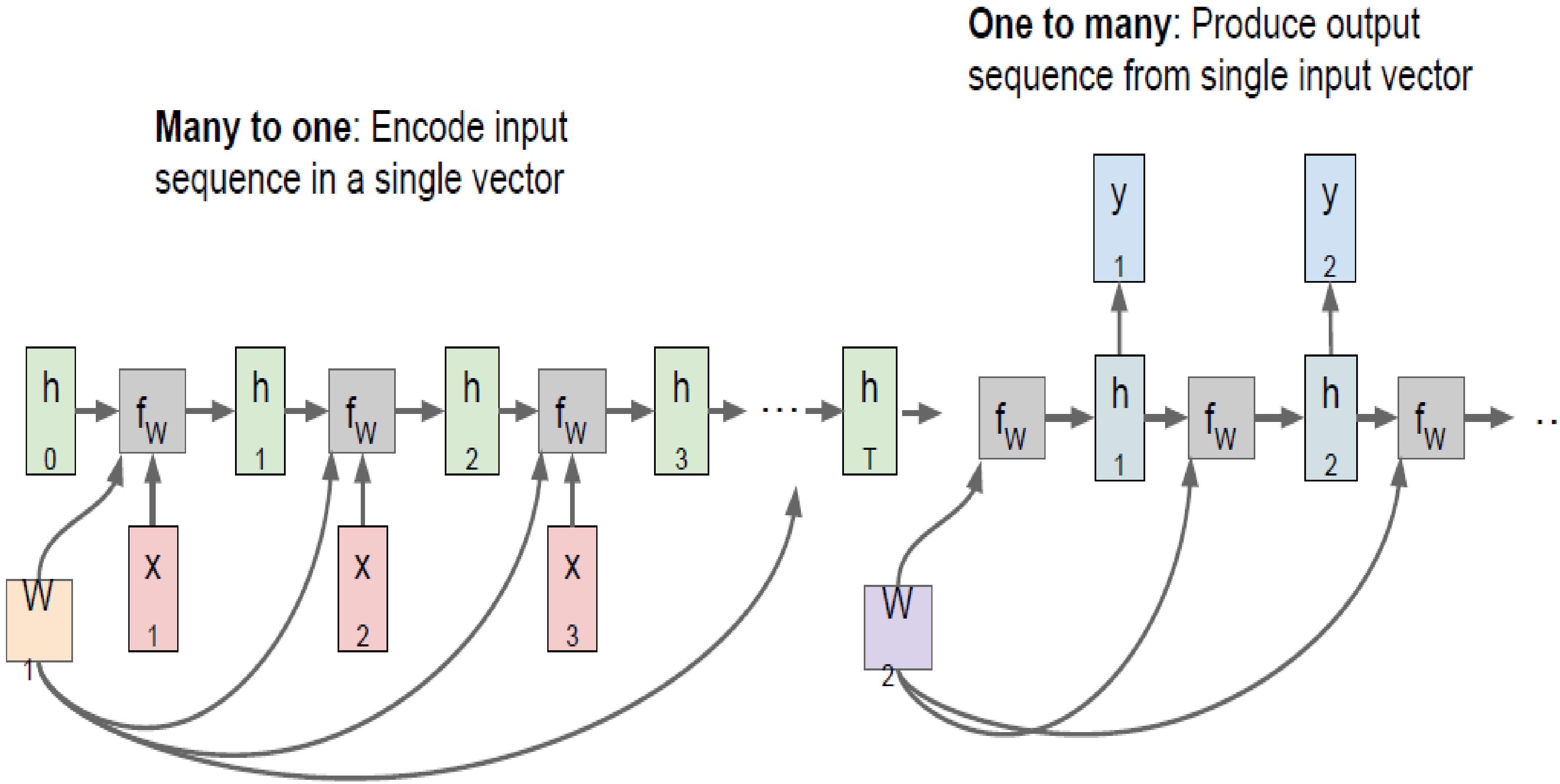
- Encoder-decoder network: a sequence-to-vector network (encoder), followed by a vector-to-sequence network (decoder).
- E.g., translating a sentence from one language to another.
- You would feed the network a sentence in one language, the encoder would convert this sentence into a single vector representation, and then the decoder would decode this vector into a sentence in another language.



Machine Translation

seq of words -> seq of words

Many-to-one +one-to-many



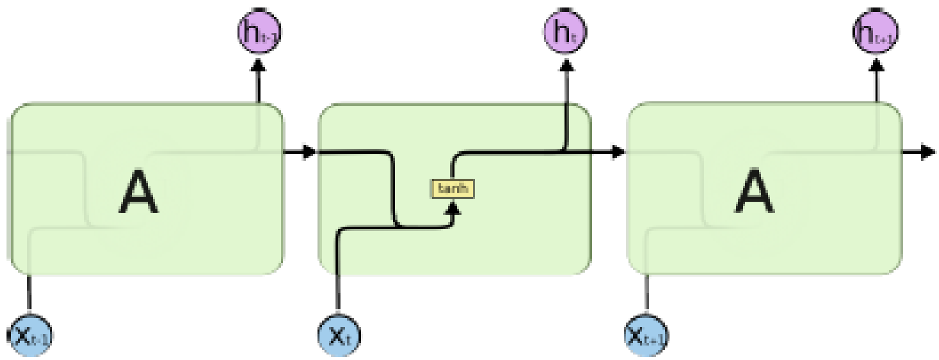
long short-term memory (LSTM)

RNN Problems

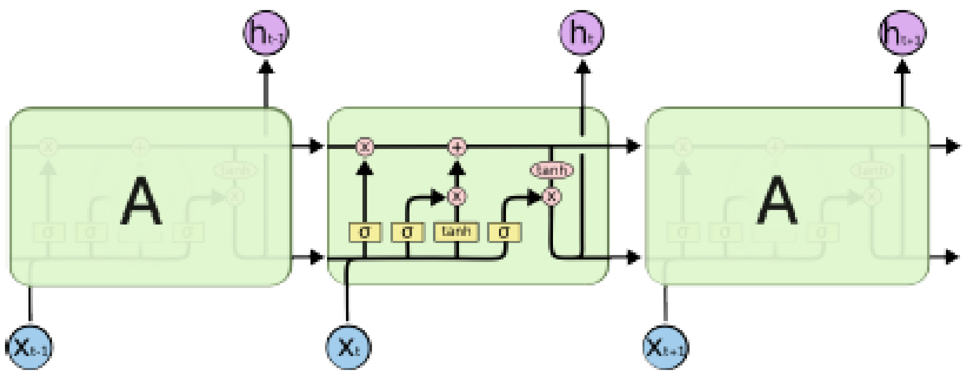
- Sometimes we only need to look at recent information to perform the present task.
  - E.g., predicting the next word based on the previous ones.
- In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.
- But, as that gap grows, RNNs become unable to learn to connect the information.
- RNNs may suffer from the vanishing/exploding gradients problem.
- To solve these problem, long short-term memory (LSTM) have been introduced.
- In LSTM, the network can learn what to store and what to throw away.

RNN Basic Cell vs. LSTM

- Without looking inside the box, the LSTM cell looks exactly like a basic cell.
- The repeating module in a standard RNN contains a single layer.

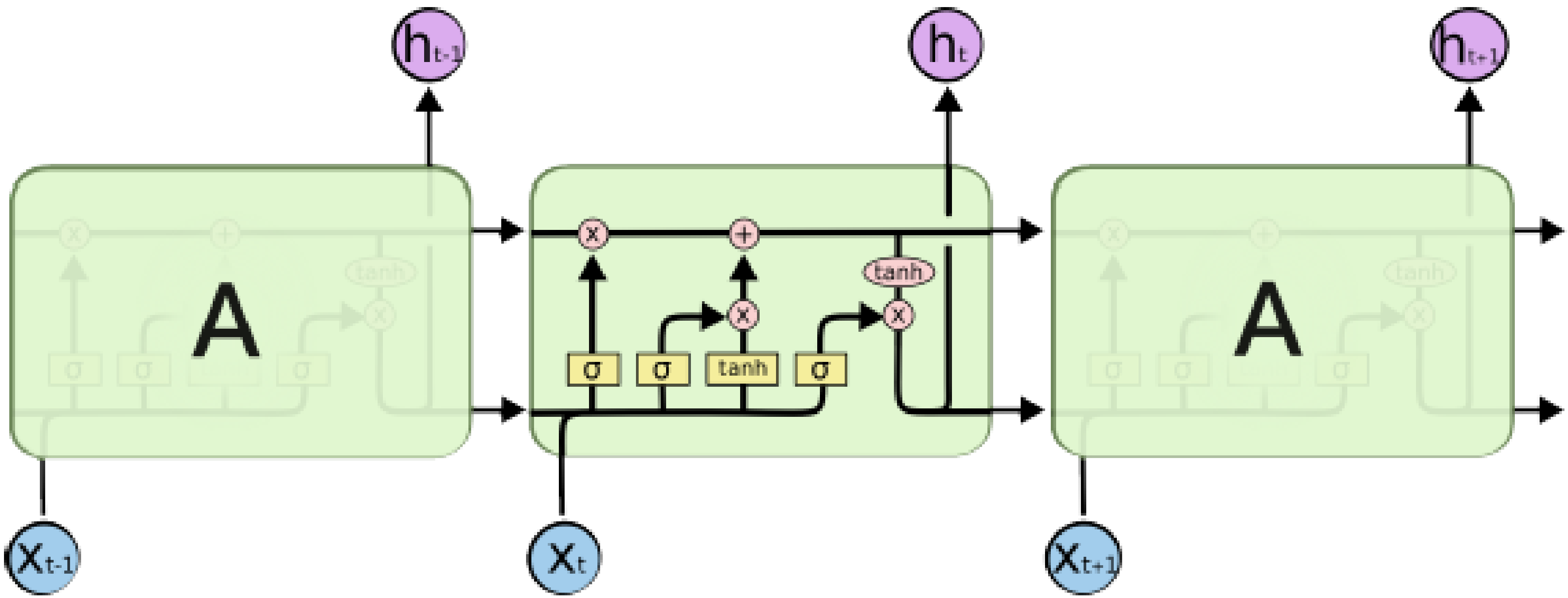


- The repeating module in an LSTM contains four interacting layers.



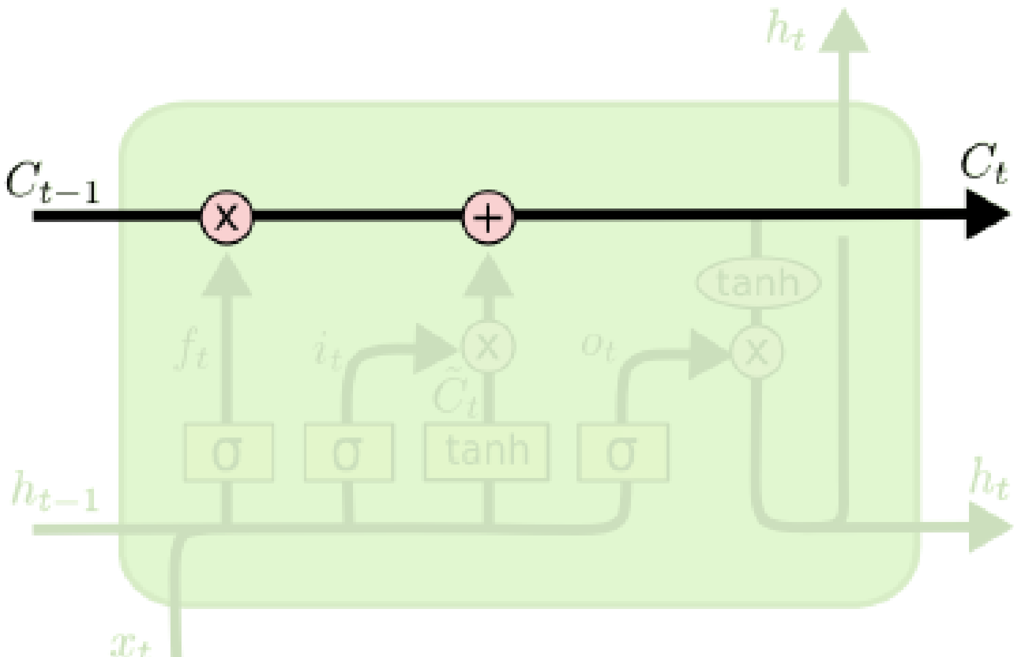
LSTM

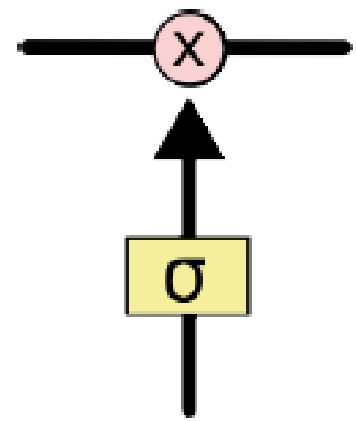
- In LSTM state is split in two vectors:
  1. $\mathtt{h}^{(t)}$ ($\mathtt{h}$ stands for hidden): the short-term state
  2. $\mathtt{c}^{(t)}$ ($\mathtt{c}$ stands for cell): the long-term state



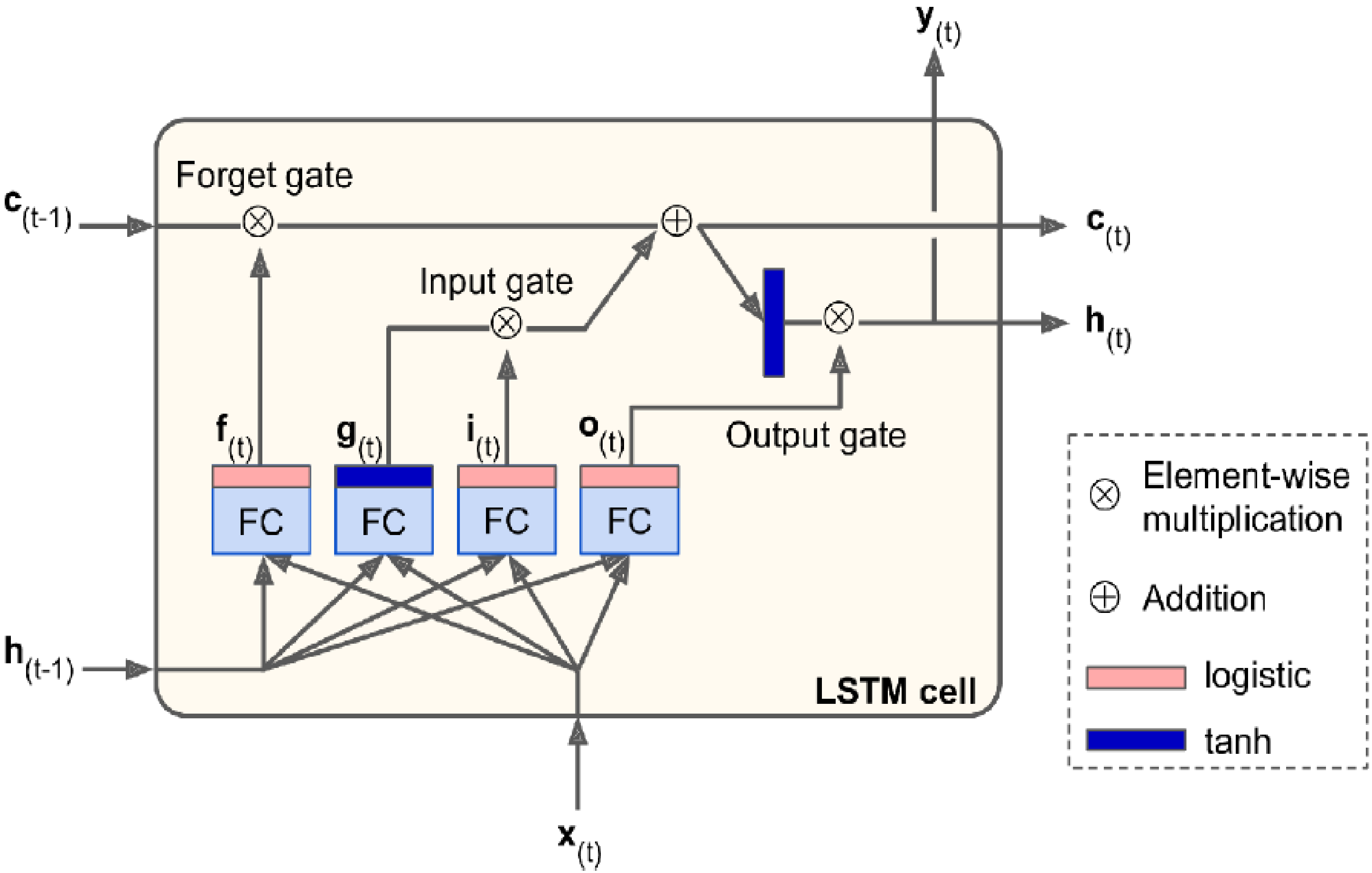
LSTM

- The cell state (long-term state), the horizontal line on the top of the diagram.
- The LSTM can remove/add information to the cell state, regulated by three gates.
  - Forget gate, input gate and output gate





LSTM

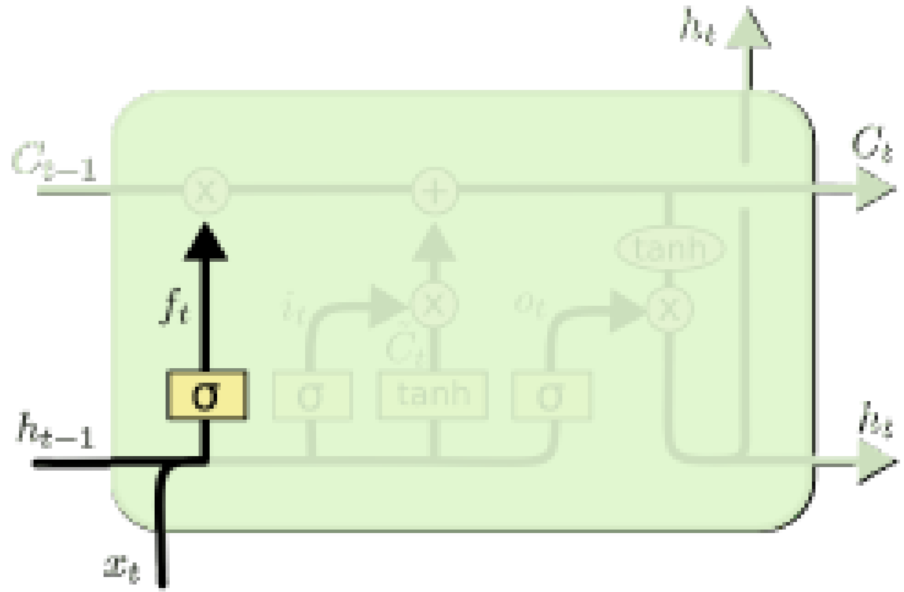


$\mathbf{y}_{(t)}$
Forget gate
$\mathbf{c}_{(t-1)}$
$\mathbf{c}_{(t)}$
Input gate
$\mathbf{h}_{(t)}$
$\mathbf{f}_{(t)}$
$\mathbf{g}_{(t)}$
$\mathbf{i}_{(t)}$
$\mathbf{o}_{(t)}$
Output gate
FC
FC
FC
FC
$\mathbf{h}_{(t-1)}$
LSTM cell
$\mathbf{x}_{(t)}$
Element-wise multiplication
Addition
logistic
tanh

Step-by-Step LSTM Walk Through

- Step one: decides what information we are going to throw away from the cell state.
- This decision is made by a sigmoid layer, called the forget gate layer.
- It looks at $\mathbf{h}^{(t-1)}$ and $\mathbf{x}^{(t)}$, and outputs a number between 0 and 1 for each number in the cell state $\mathbf{c}^{(t-1)}$.
  - 1 represents completely keep this, and 0 represents completely get rid of this.

$$\mathbf{f}^{(t)} = \sigma(\mathbf{u}_f^T \mathbf{x}^{(t)} + \mathbf{w}_f \mathbf{h}^{(t-1)})$$

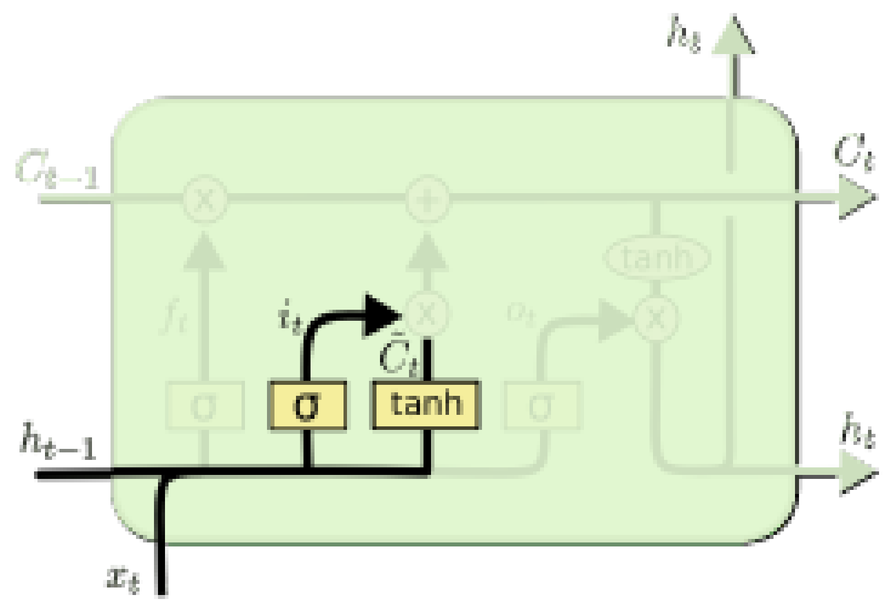


Step-by-Step LSTM Walk Through

- Second step: decides what new information we are going to store in the cell state. This has two parts:
- 1. A sigmoid layer, called the input gate layer, decides which values we will update.
- 2. A tanh layer creates a vector of new candidate values that could be added to the state.

$$\mathbf{i}^{(t)} = \sigma(\mathbf{u}_i^T \mathbf{x}^{(t)} + \mathbf{w}_i \mathbf{h}^{(t-1)})$$

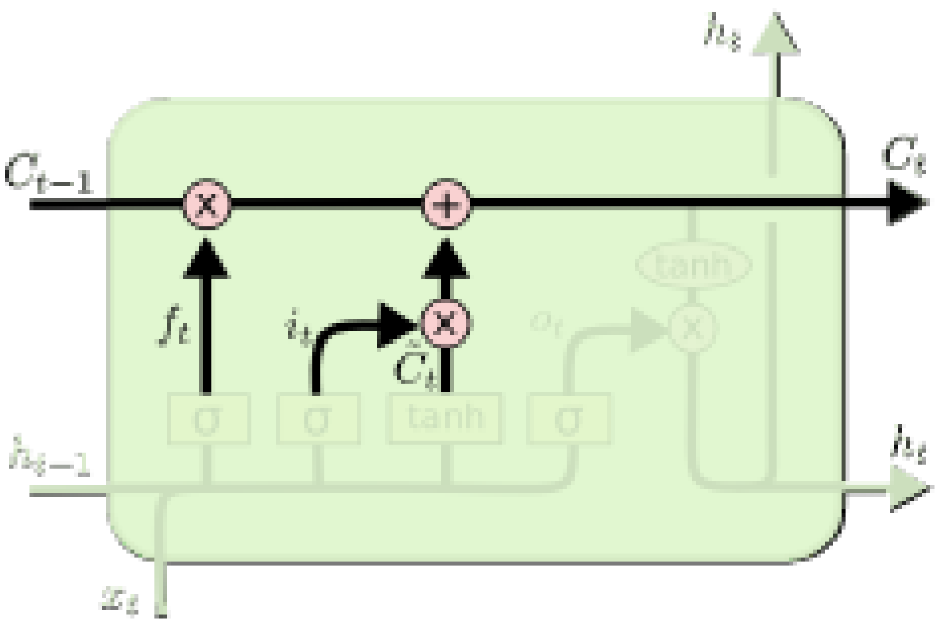
$$\tilde{\mathbf{c}}^{(t)} = \tanh(\mathbf{u}_{\tilde{c}}^T \mathbf{x}^{(t)} + \mathbf{w}_{\tilde{c}} \mathbf{h}^{(t-1)})$$



Step-by-Step LSTM Walk Through

- Third step: updates the old cell state $\mathtt{c}^{(t-1)}$, into the new cell state $\mathtt{c}^{(t)}$.
- We multiply the old state by $\mathtt{f}^{(t)}$, forgetting the things we decided to forget earlier.
- Then we add it $\mathtt{i}^{(t)} \otimes \tilde{\mathtt{c}}^{(t)}$.
- This is the new candidate values, scaled by how much we decided to update each state value.

$$\mathtt{c}^{(t)} = \mathtt{f}^{(t)} \otimes \mathtt{c}^{(t-1)} + \mathtt{i}^{(t)} \otimes \tilde{\mathtt{c}}^{(t)}$$

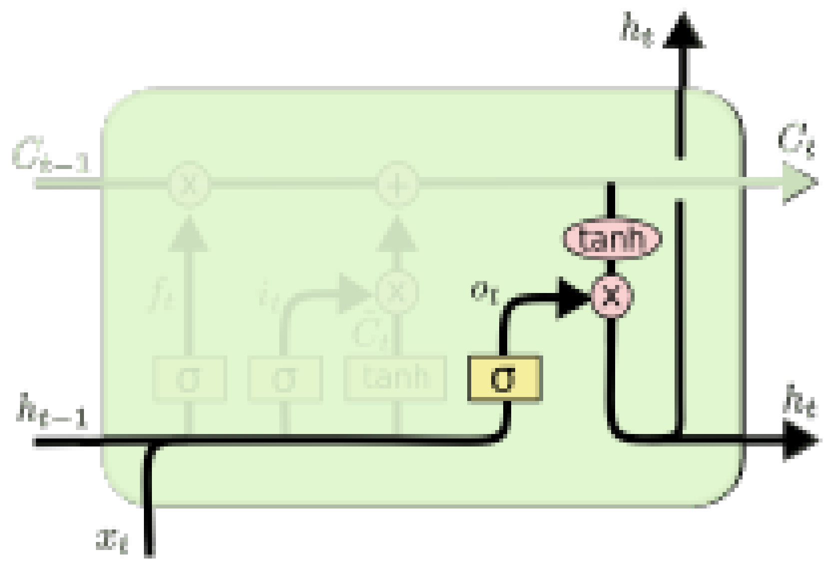


Step-by-Step LSTM Walk Through

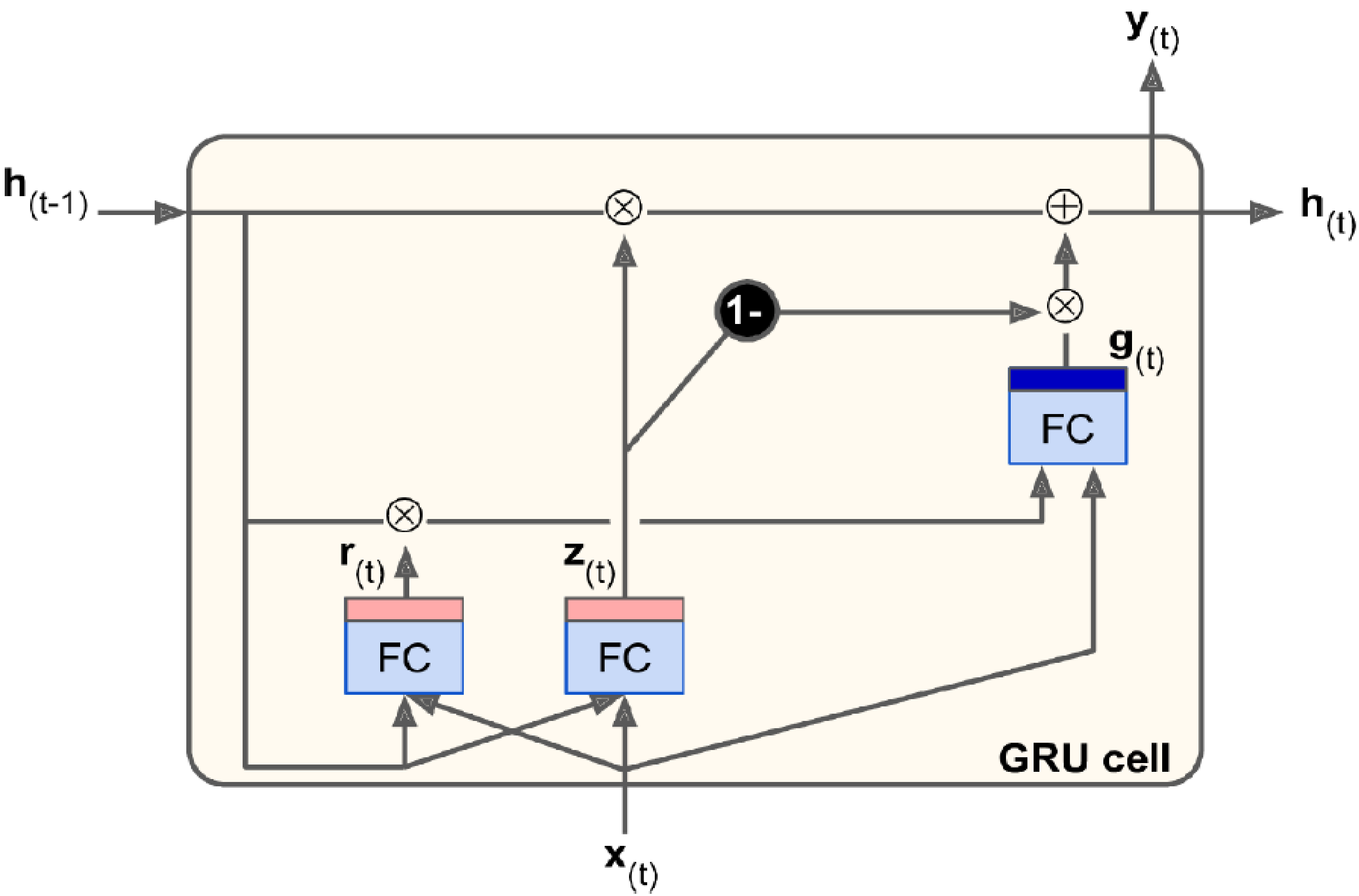
- Fourth step: decides about the output.
- First, runs a sigmoid layer that decides what parts of the cell state we are going to output.
- Then, puts the cell state through tanh and multiplies it by the output of the sigmoid gate (output gate), so that it only outputs the parts it decided to.

$$o^{(t)} = \sigma(u_o^T x^{(t)} + w_o h^{(t-1)})$$

$$\hat{y}^{(t)} = h^{(t)} = o^{(t)} \otimes \tanh(c^{(t)})$$



Gated Recurrent Unit (GRU)



Gated Recurrent Unit (GRU)

$$\mathbf{z}_{(t)} = \sigma\left(\mathbf{W}_{xz}{}^{\mathsf{T}}\mathbf{x}_{(t)} + \mathbf{W}_{hz}{}^{\mathsf{T}}\mathbf{h}_{(t-1)} + \mathbf{b}_{z}\right)$$

$$\mathbf{r}_{(t)} = \sigma\left(\mathbf{W}_{xr}{}^{\mathsf{T}}\mathbf{x}_{(t)} + \mathbf{W}_{hr}{}^{\mathsf{T}}\mathbf{h}_{(t-1)} + \mathbf{b}_{r}\right)$$

$$\mathbf{g}_{(t)} = \tanh\left(\mathbf{W}_{xg}{}^{\mathsf{T}}\mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{\mathsf{T}}\left(\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}\right) + \mathbf{b}_{g}\right)$$

$$\mathbf{h}_{(t)} = \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + \left(1 - \mathbf{z}_{(t)}\right) \otimes \mathbf{g}_{(t)}$$

WaveNet- Use 1D convolutional layers