# CS 470 Deep Learning Practice

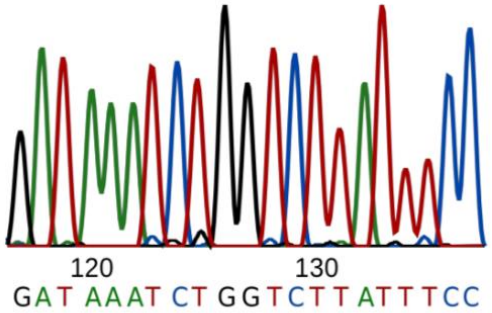
# Day 5

Yechan Hwang

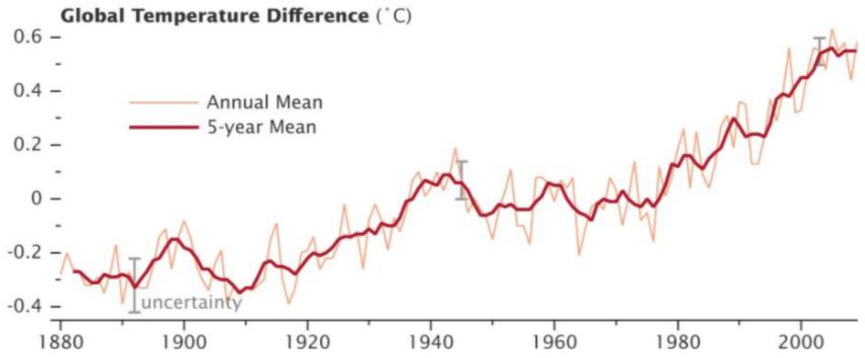
yemintmint@kaist.ac.kr

# Motivation

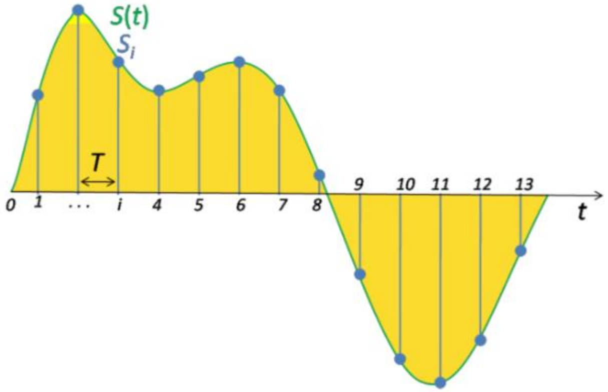
- Let's say that we want to deal with sequential data such as speech, text, audio or video




**DNA sequence data**




**Temporal sequence**

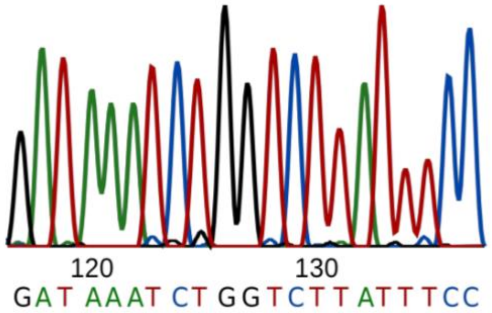



**Audio data**

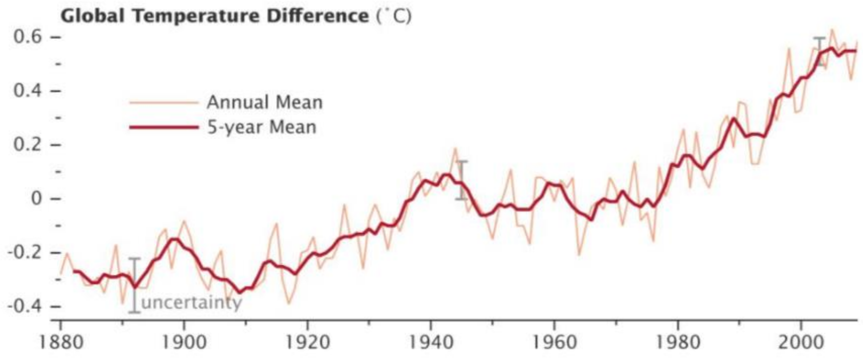
# Motivation

- Let's say that we want to deal with sequential data such as speech, text, audio or video
- Our model should be able to

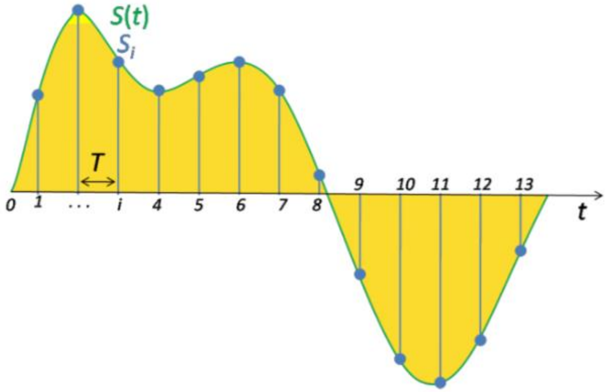
    (1) process data of arbitrary length




**DNA sequence data**




**Temporal sequence**



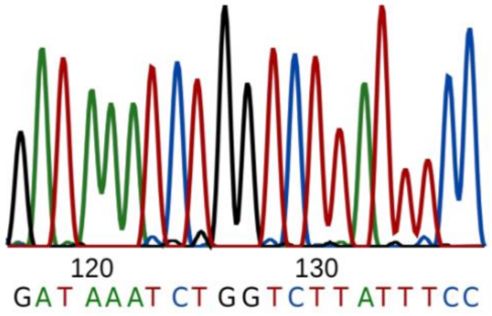

**Audio data**

# Motivation

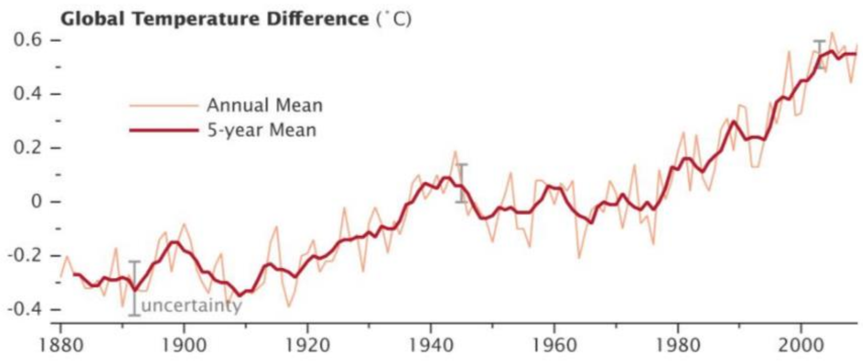
- Let's say that we want to deal with sequential data such as speech, text, audio or video
- Our model should be able to

  (1) process data of arbitrary length

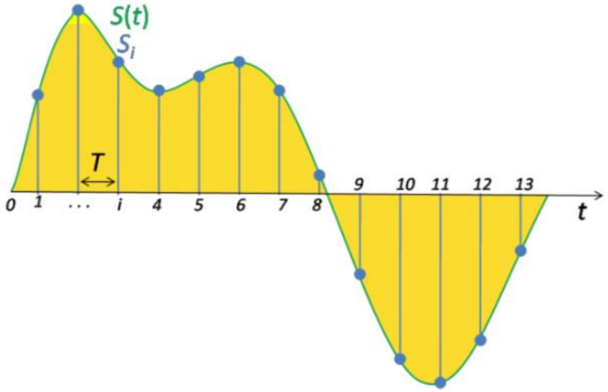
  (2) interpret the meaning contained in the order of data




**DNA sequence data**



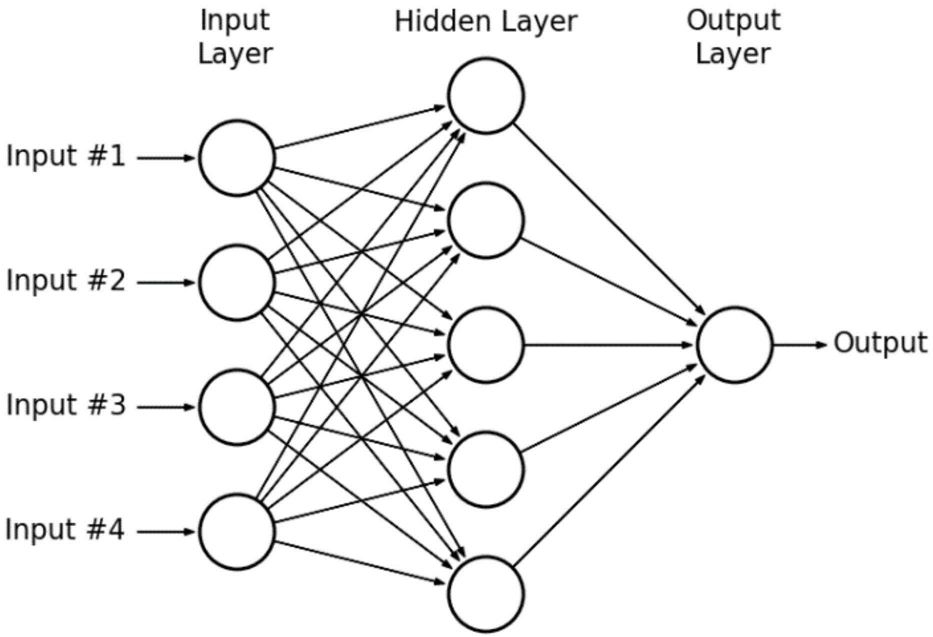

**Temporal sequence**




**Audio data**

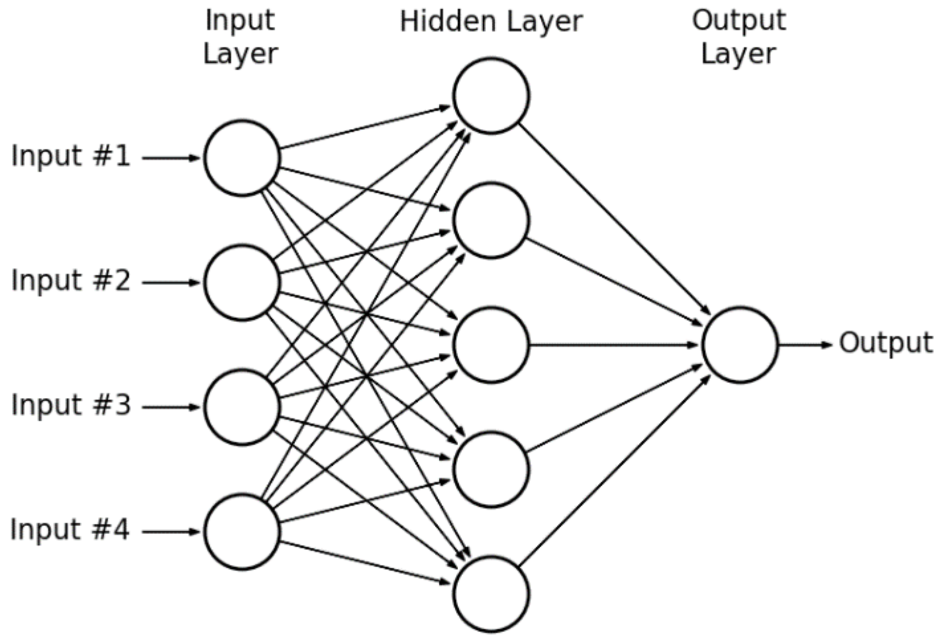
# Modeling sequences with feedforward network

- Option 1 : MLP



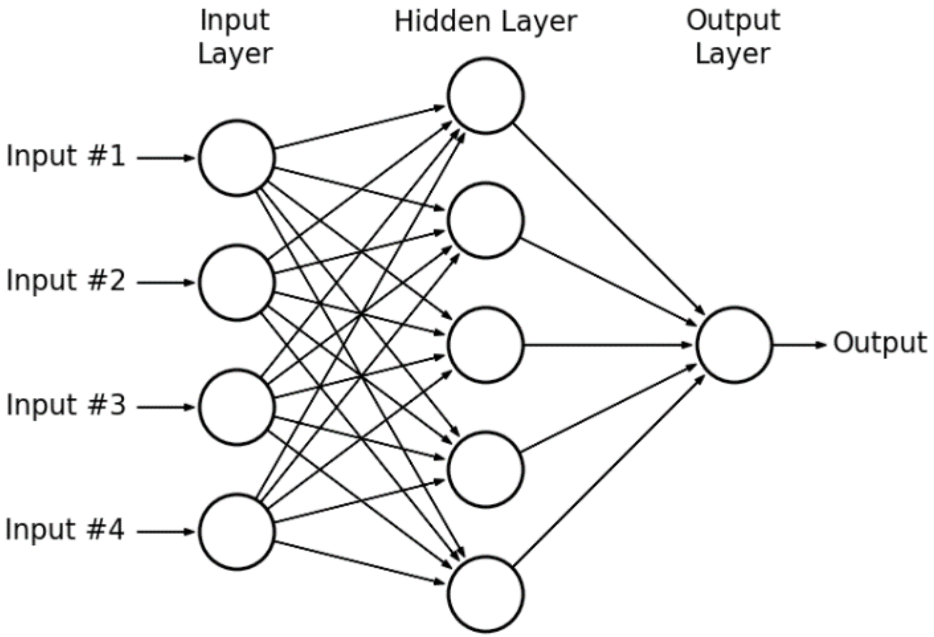
# Modeling sequences with feedforward network

- Option 1 : MLP
- What if we change the sequence length?



# Modeling sequences with feedforward network

- Option 1 : MLP
- What if we change the sequence length?
  - We cannot reuse the same network for handling sequences in different length
  - It is because the network parameter fixes the length of the sequences

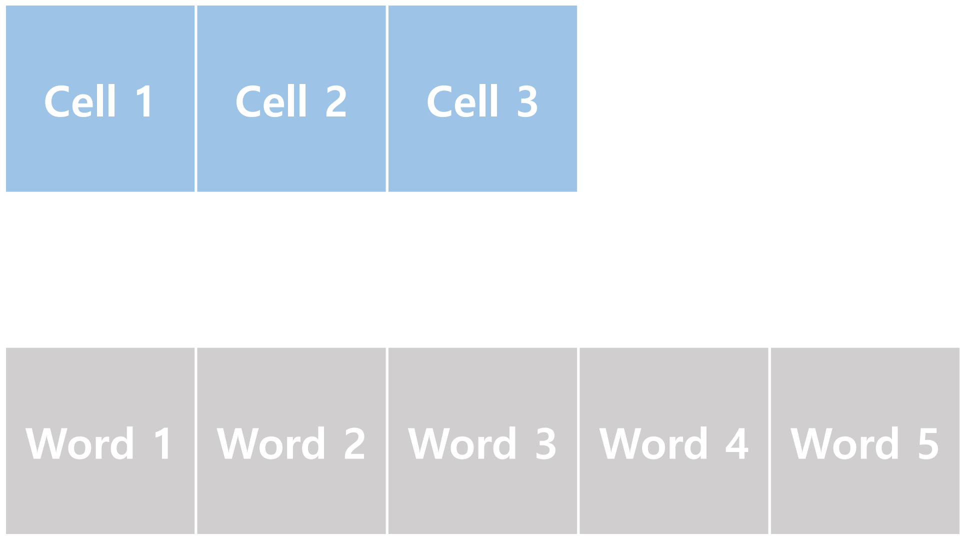


# Modeling sequences with feedforward network

- Option 2 : CNN
- What if we change the sequence length?
  - We can reuse the same network by sliding convolution filters over the sequence.

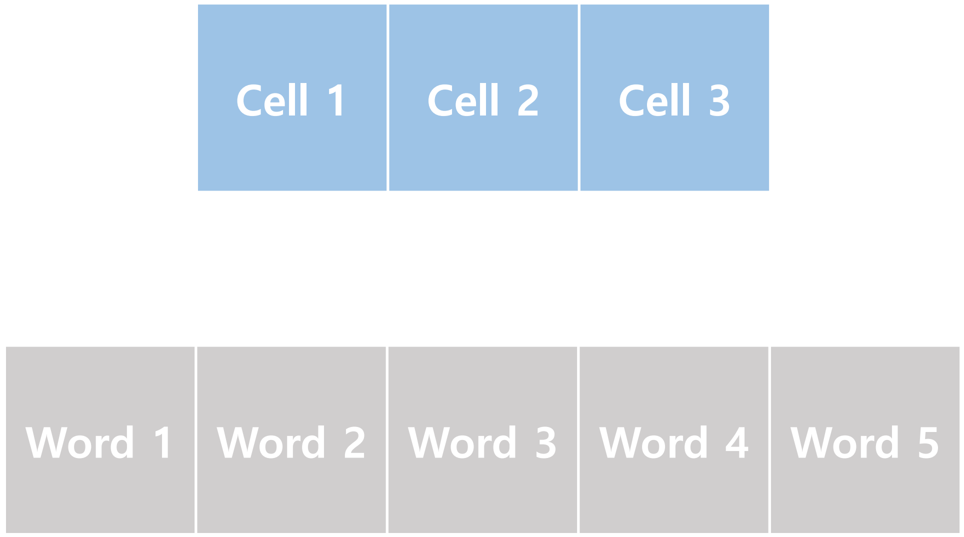
# Modeling sequences with feedforward network

- Option 2 : CNN
- What if we change the sequence length?
  - We can reuse the same network by sliding convolution filters over the sequence.



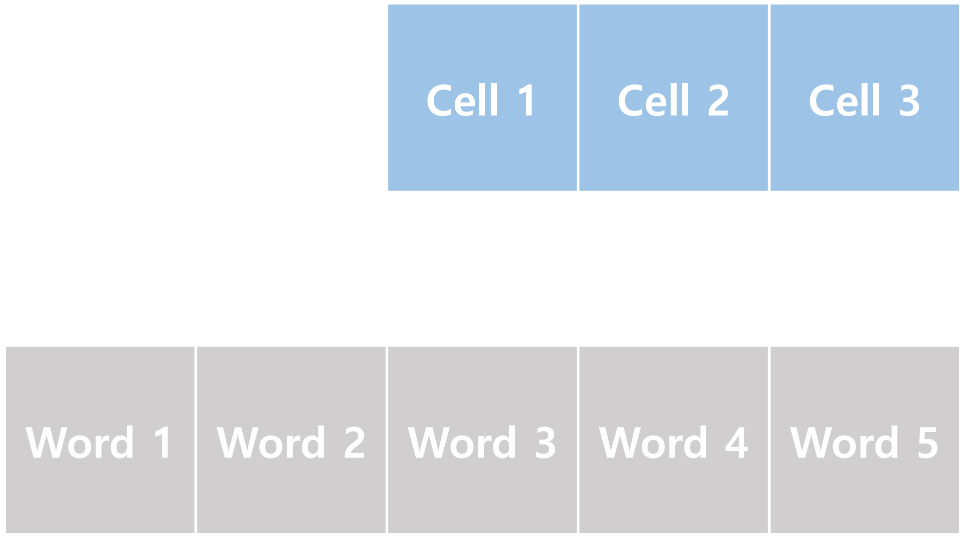
# Modeling sequences with feedforward network

- Option 2 : CNN
- What if we change the sequence length?
  - We can reuse the same network by sliding convolution filters over the sequence.



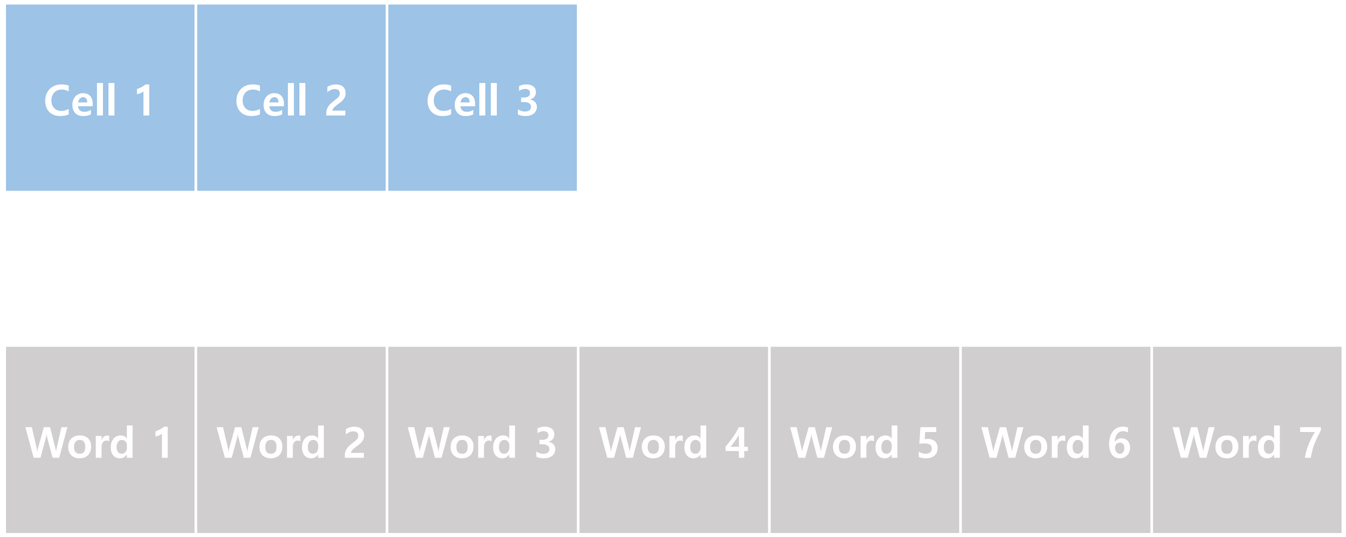
# Modeling sequences with feedforward network

- Option 2 : CNN
- What if we change the sequence length?
  - We can reuse the same network by sliding convolution filters over the sequence.



# Modeling sequences with feedforward network

- Option 2 : CNN
- What if we change the sequence length?
  - We can reuse the same network by sliding convolution filters over the sequence.



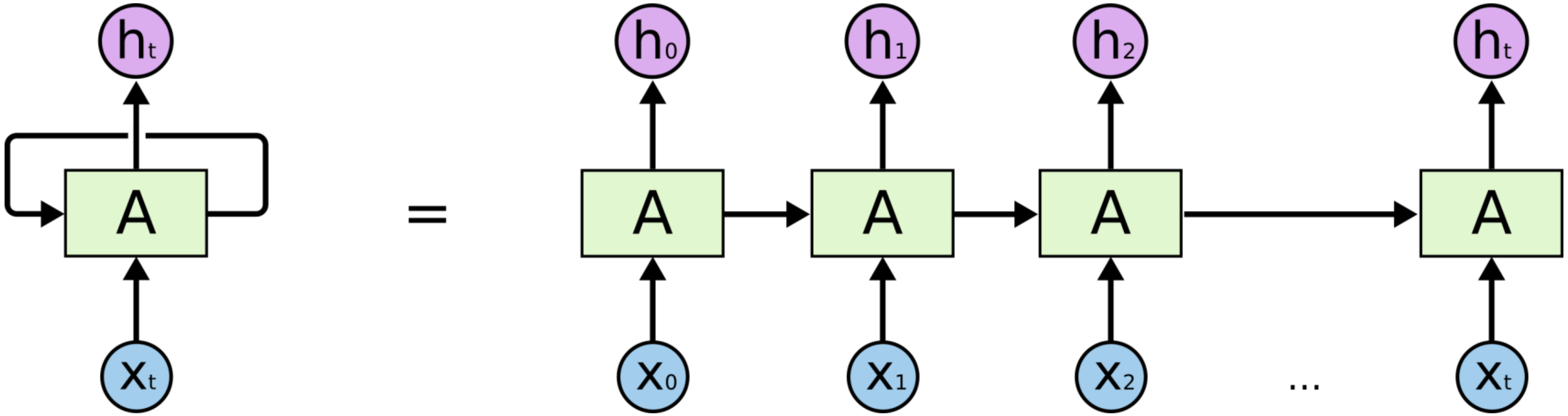
# Modeling sequences with feedforward network

- Option 2 : CNN
- What if we change the sequence length?
  - We can reuse the same network by sliding convolution filters over the sequence.

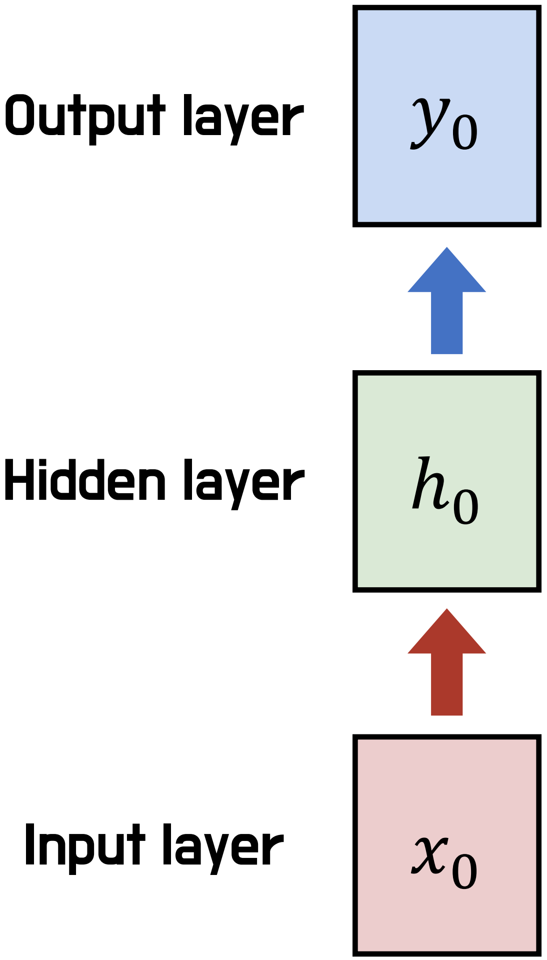
- Problems?
  - The hidden representation grows with the length of the sequence.
  - The receptive field is fixed. (More critical!)

# Recurrent Neural Network

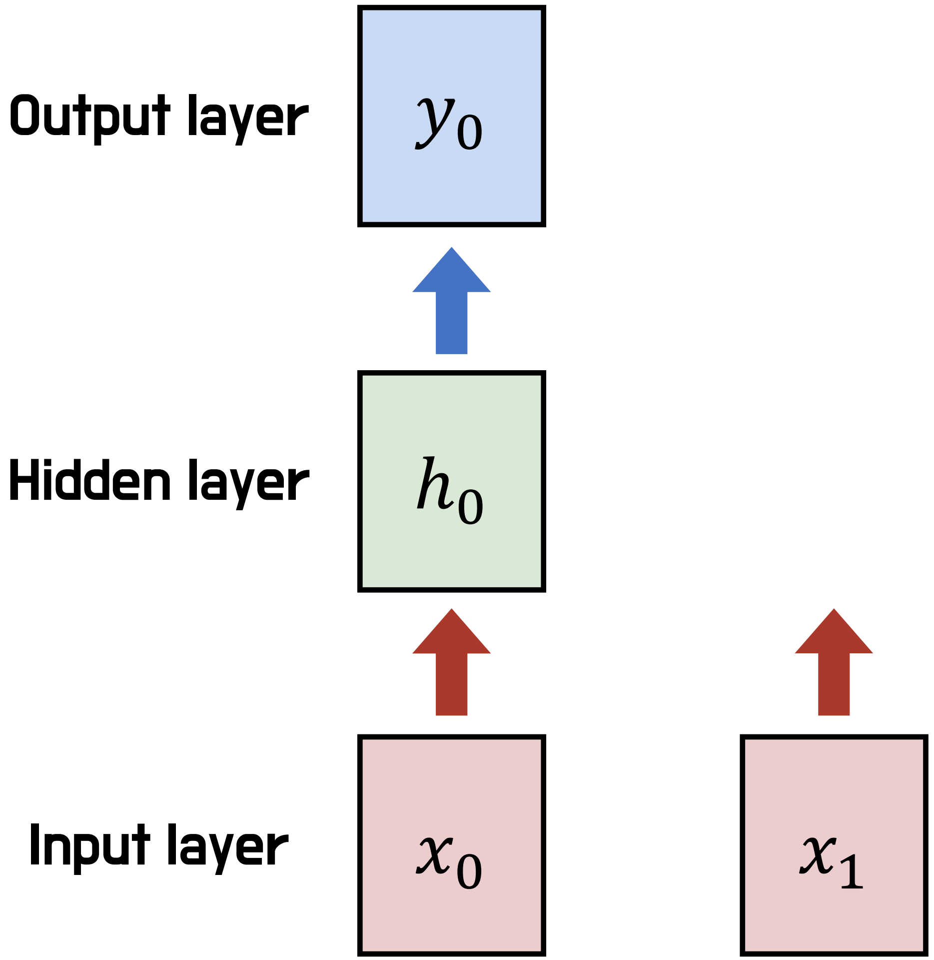
- Suitable for sequence data with important temporal dynamics
  - It can handle arbitrary input length and consider the order of time units
- It can remember its input by using its internal memory
  - The current output result is affected by the result of the previous time step



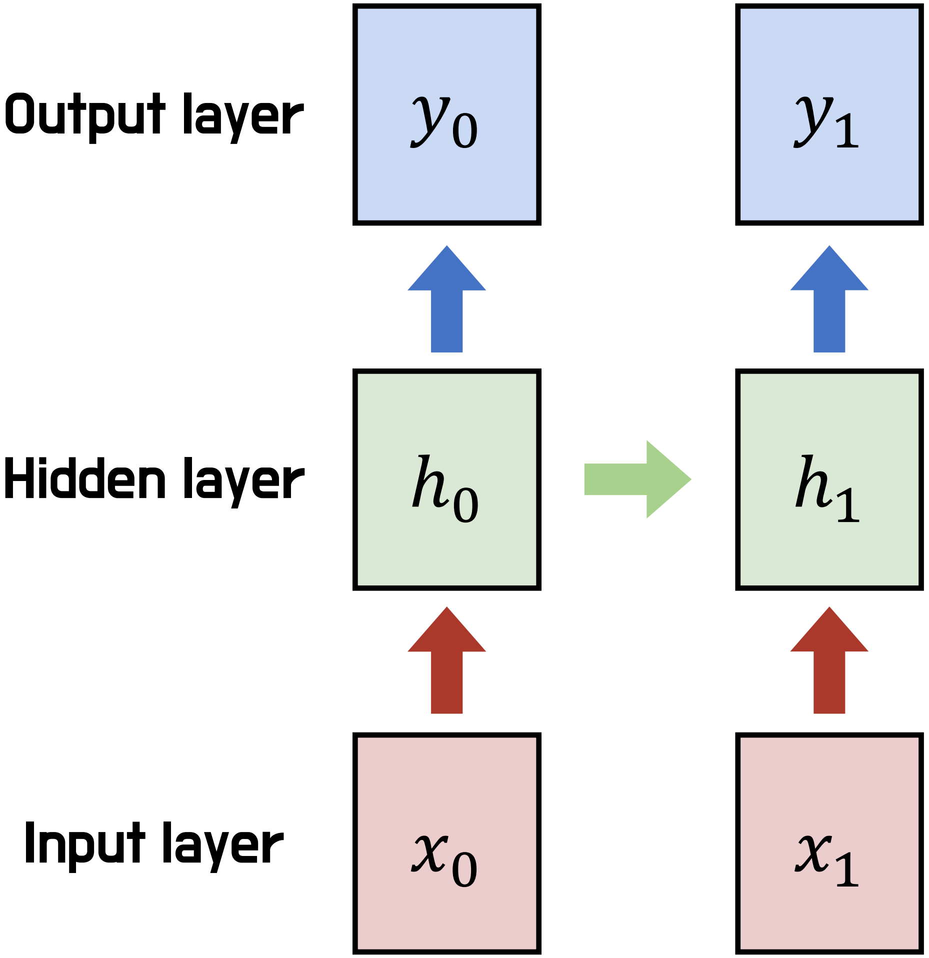
# Recurrent Neural Network



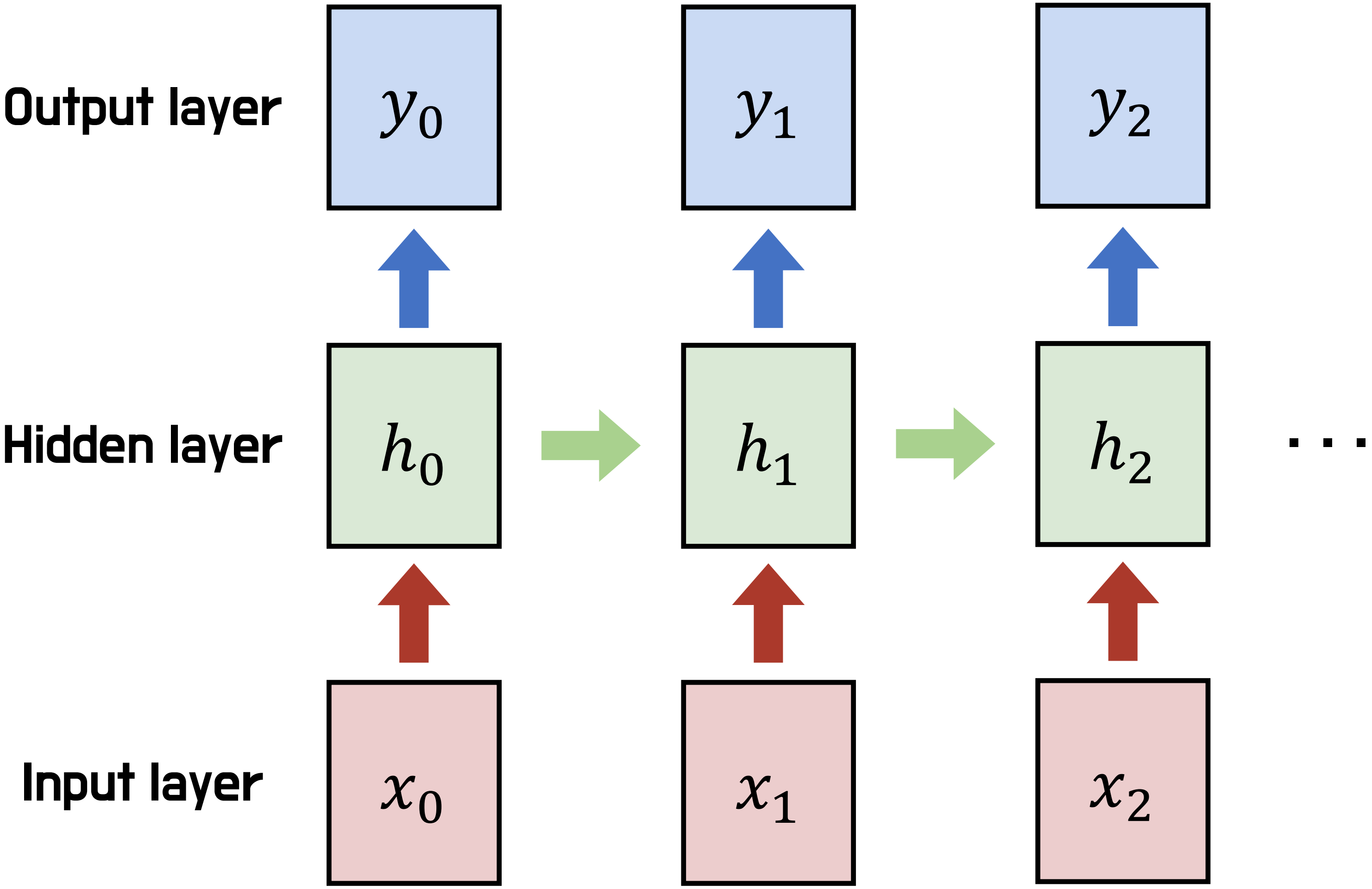
# Recurrent Neural Network



# Recurrent Neural Network

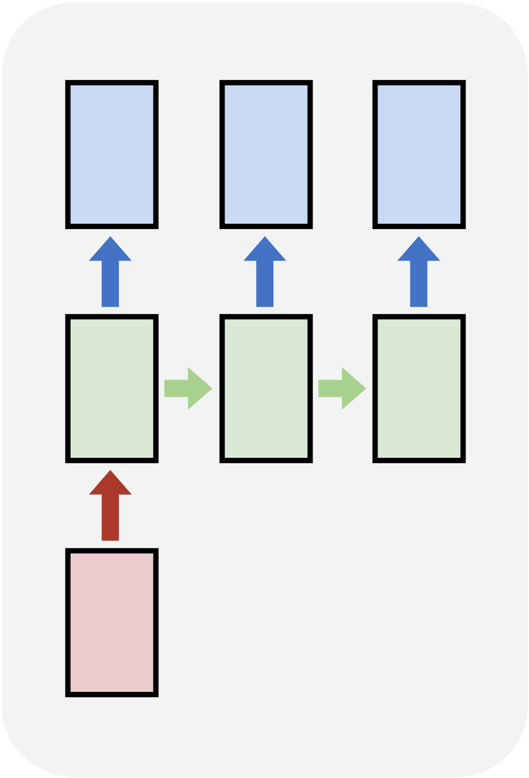


# Recurrent Neural Network

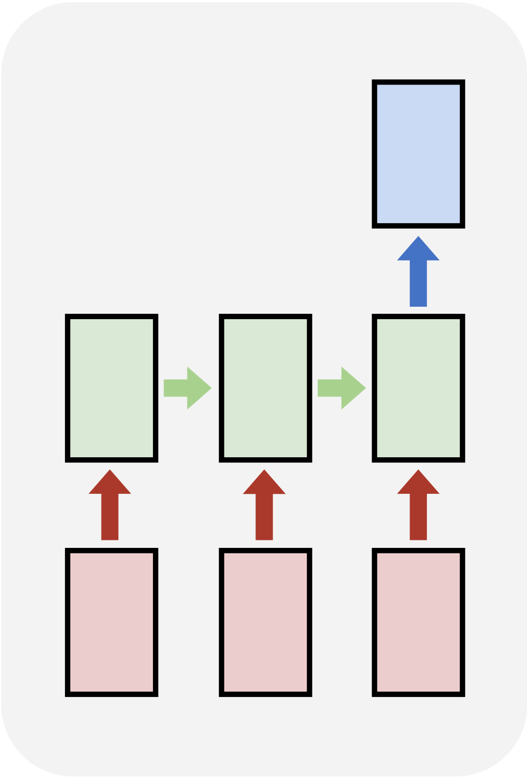


# Types of RNNs

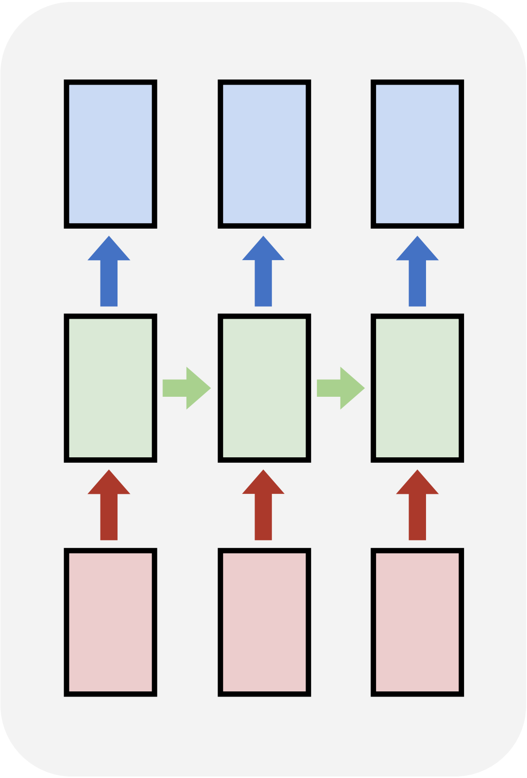
**One to many**



**Many to one**

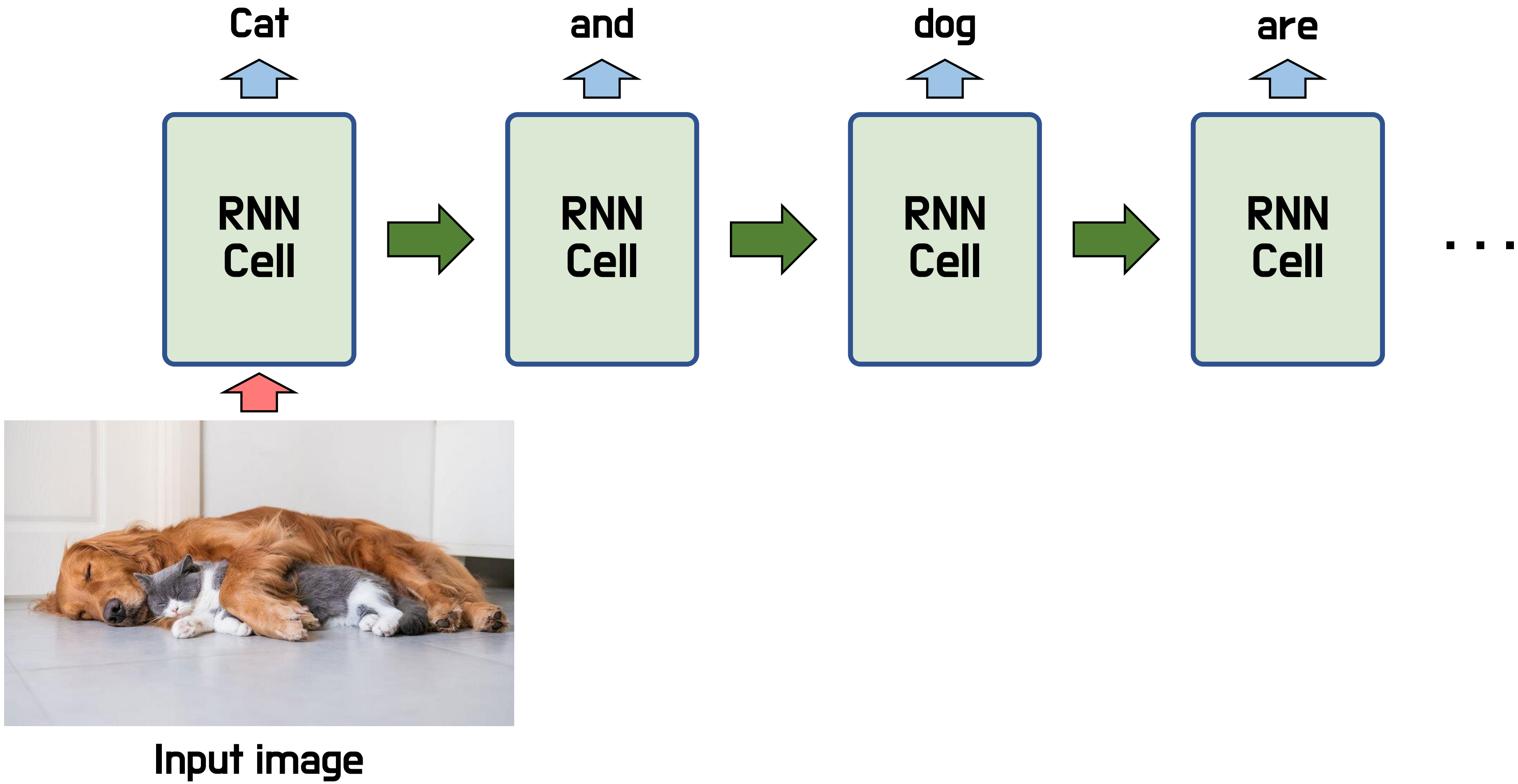


**Many to many**



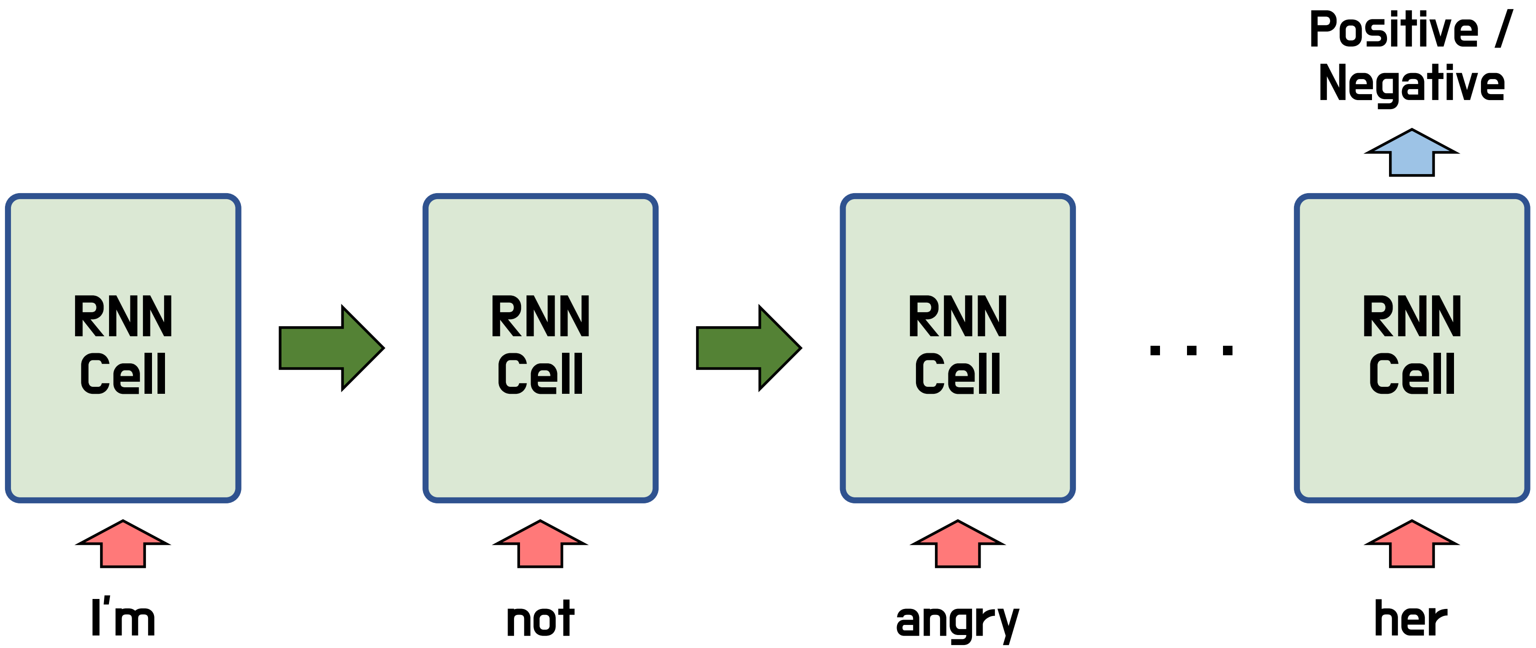
# Types of RNNs

- One to many model
- Example : Generating sentence that explains the input image



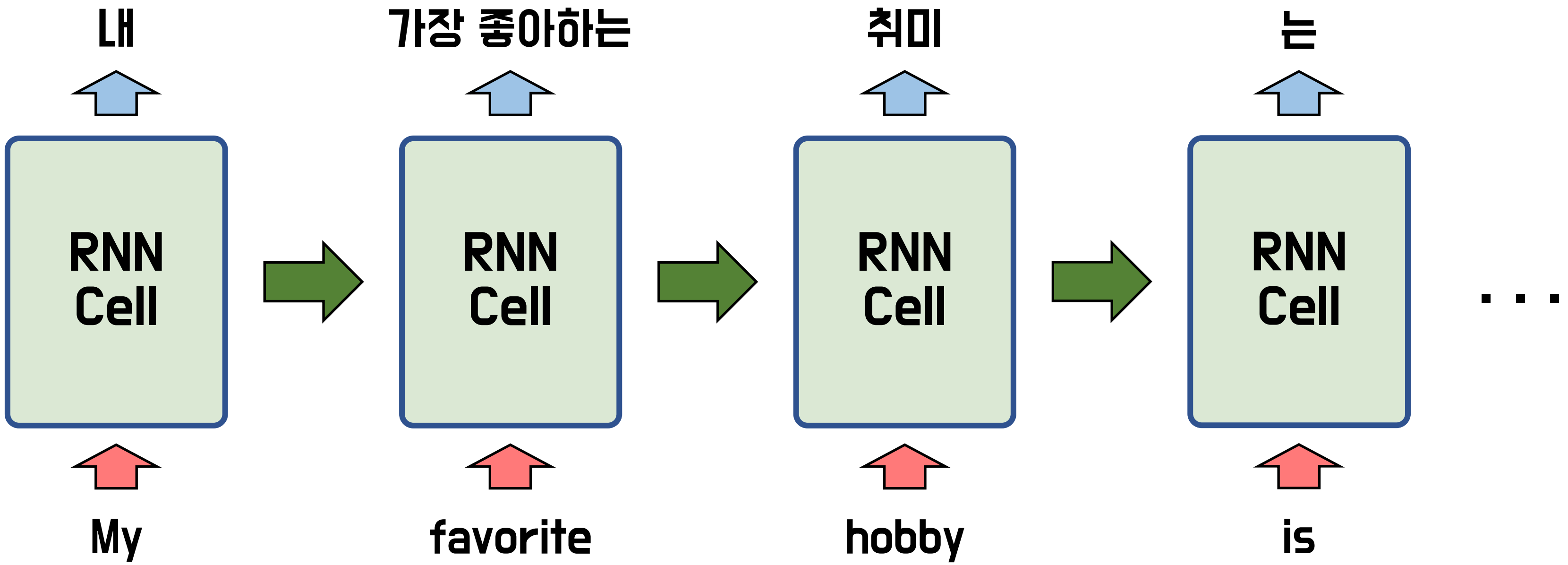
# Types of RNNs

- Many to one model
- Example : Sentence sentiment analysis



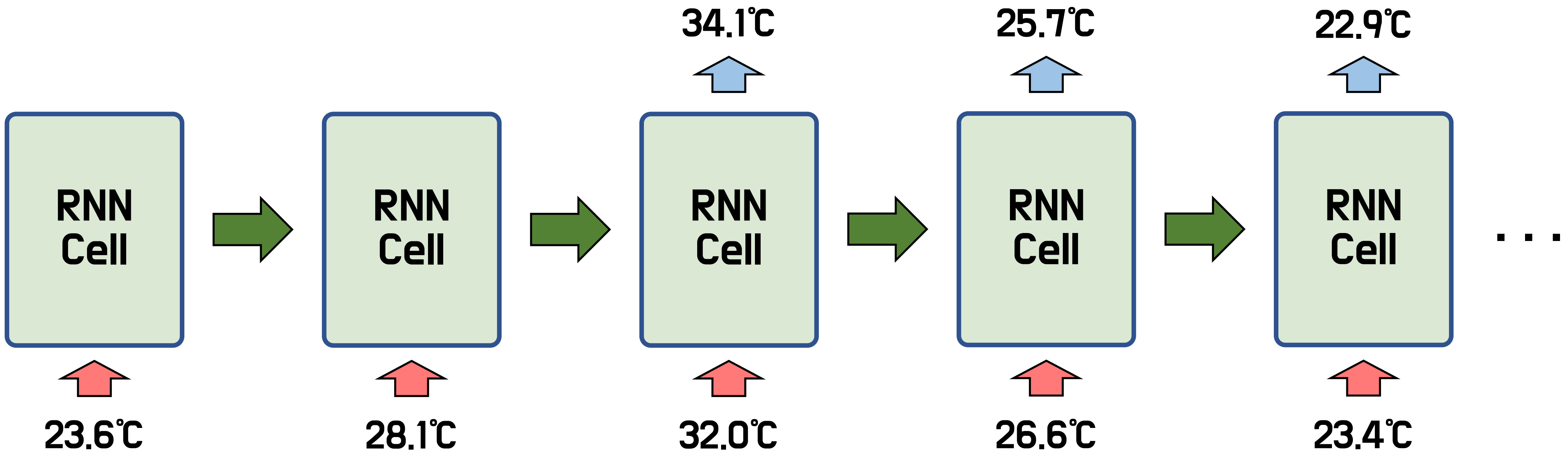
# Types of RNNs

- Many to many model
- Example : Translating sentences



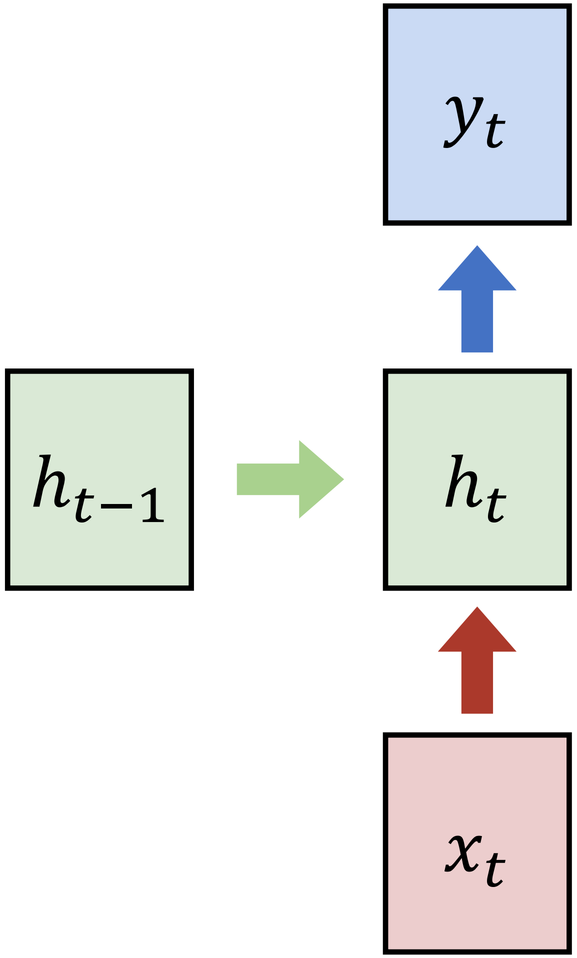
# Types of RNNs

- Many to many model
- Example : Translating sentences
- Don’t have to make output every time



# Recurrent Neural Network

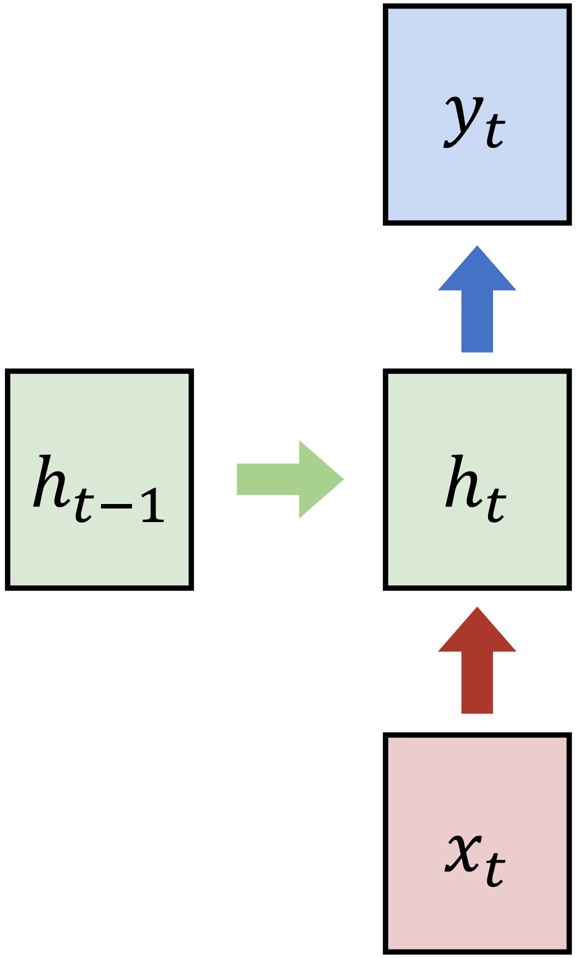
- How to calculate the hidden state of RNNs




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

# Recurrent Neural Network

- How to calculate the hidden state of RNNs




previous
hidden state

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

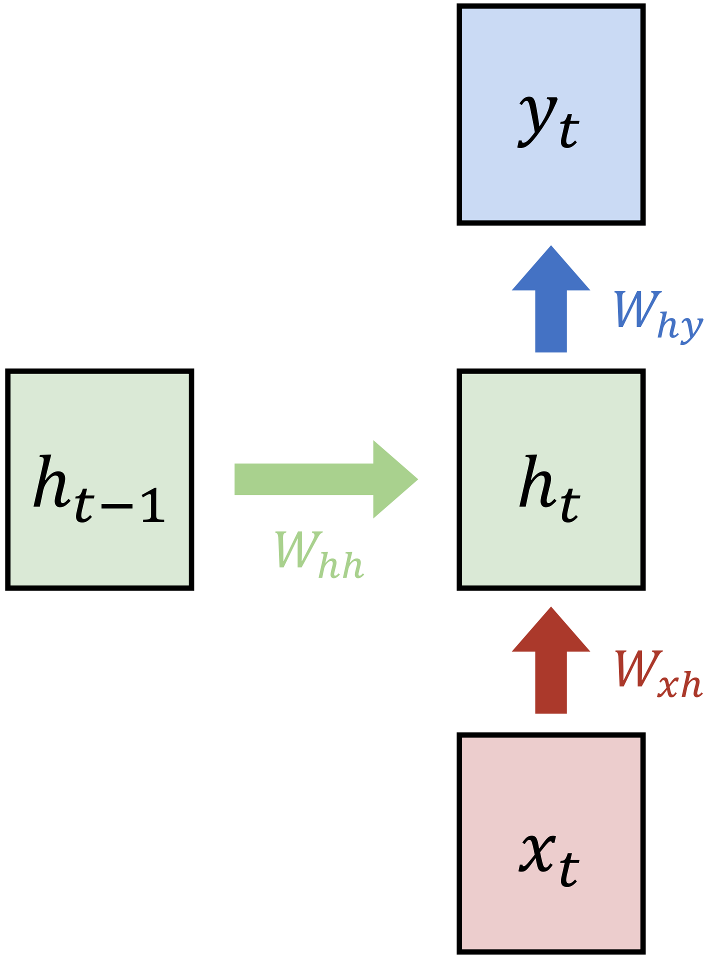
new hidden
state

input vector
for now

some function with
parameters W

# Recurrent Neural Network

- Weights for RNN



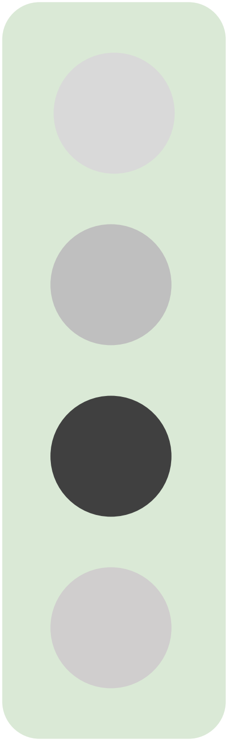

$$y_t = W_{hy} h_t + b_y$$

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

Notice : all the weights of RNN ($W_{hh}$, $b_h$, $W_{xh}$, $W_{hy}$, $b_y$) are shared across all time steps

# Activation function for RNN

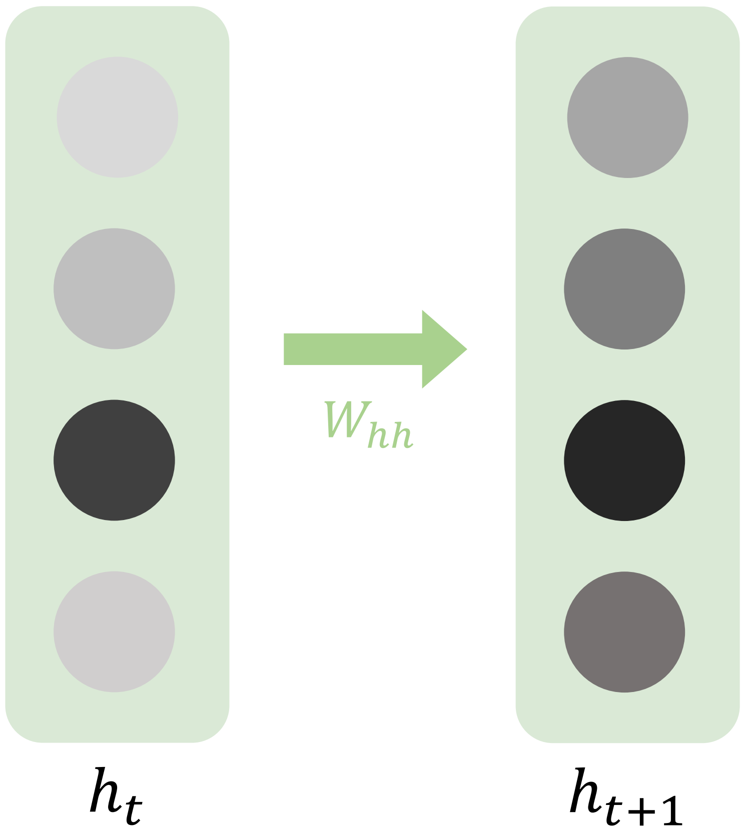
- Let's assume that some hidden state has a large node value at early time step and we use ReLU as an activation function.



$h_t$

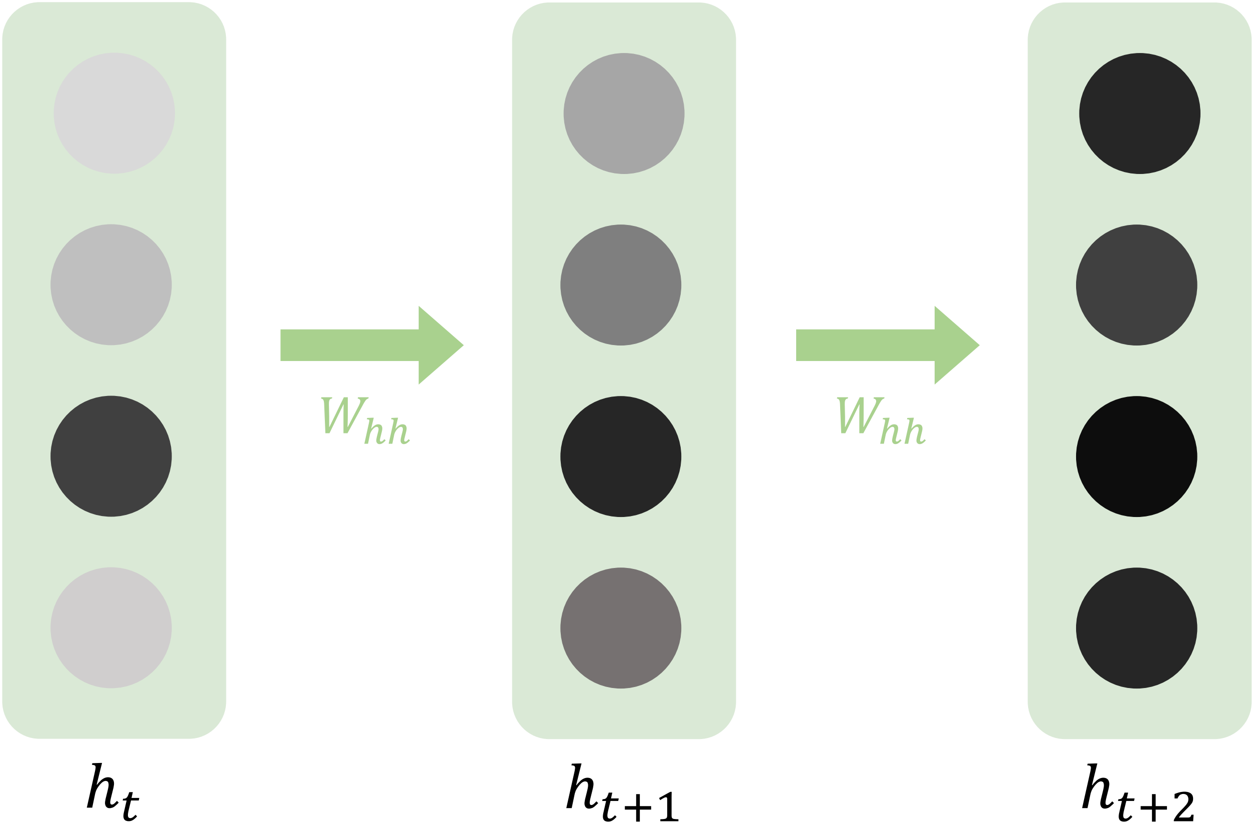
# Activation function for RNN

- Let's assume that some hidden state has a large node value at early time step and we use ReLU as an activation function.
- Then at the next time step, this large value is again multiplied by $W_{hh}$ and can output larger value.



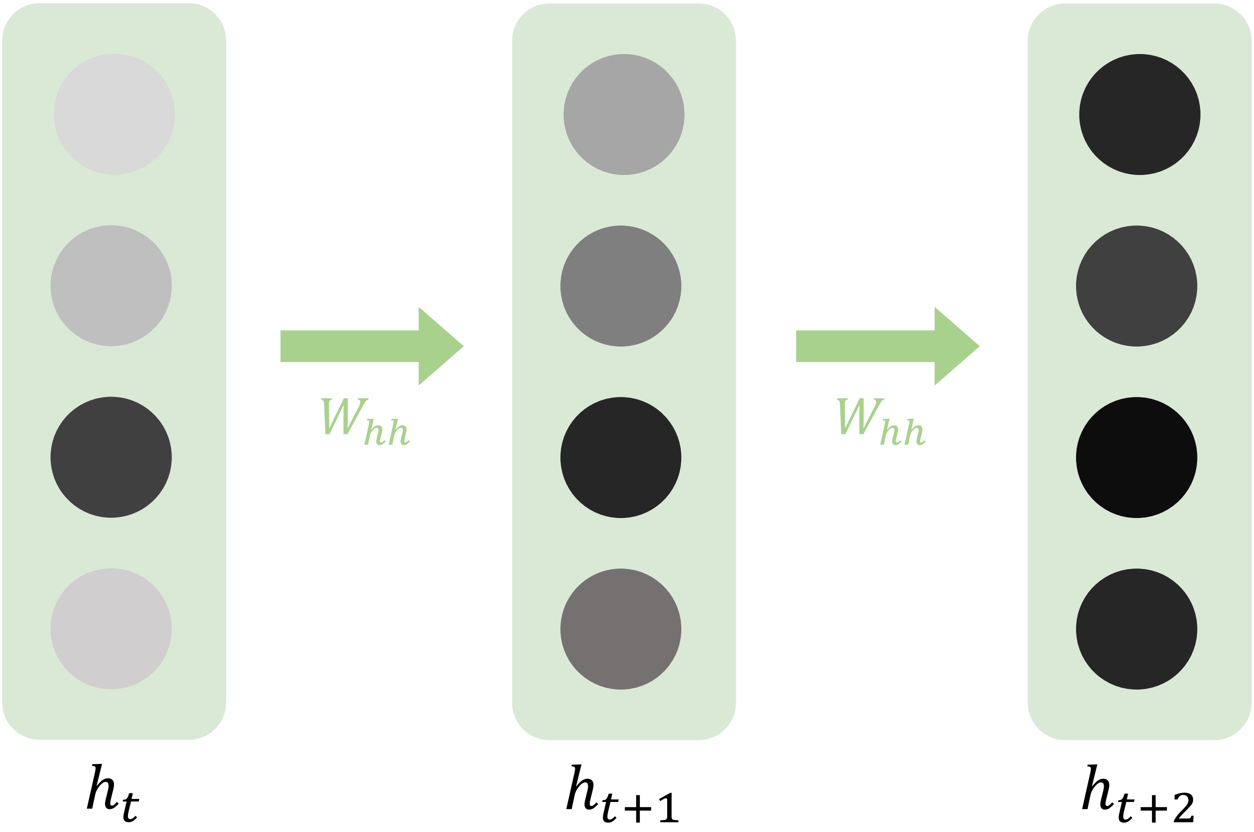
# Activation function for RNN

- Let's assume that some hidden state has a large node value at early time step and we use ReLU as an activation function.
- Then at the next time step, this large value is again multiplied by $W_{hh}$ and can output larger value.



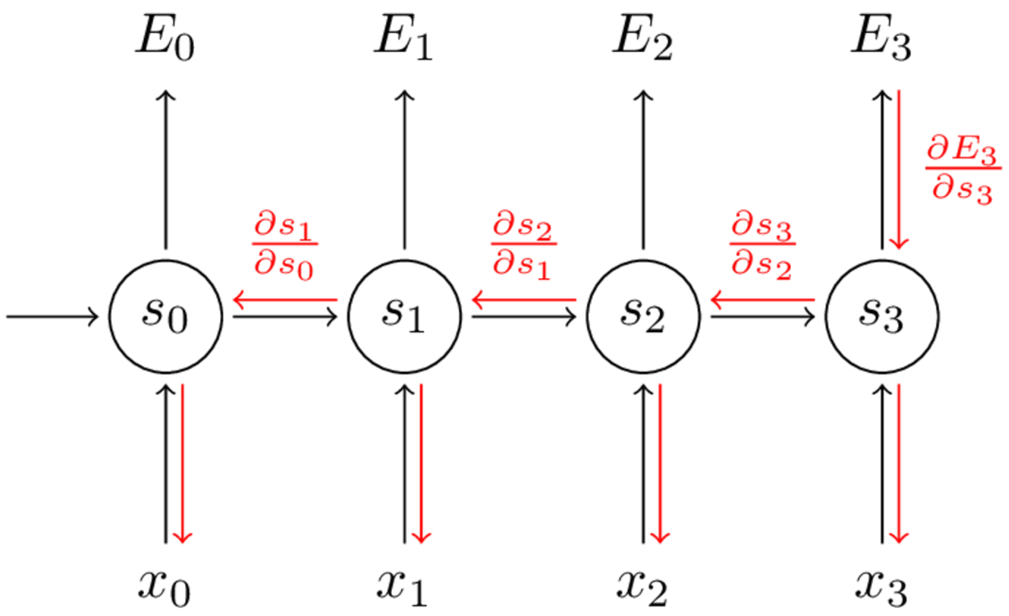
# Activation function for RNN

- Let's assume that some hidden state has a large node value at early time step and we use ReLU as an activation function.
- Then at the next time step, this large value is again multiplied by $W_{hh}$ and can output larger value.
- Therefore, if we use ReLU in RNN, activation value can explode (use *tanh* or *sigmoid* instead).



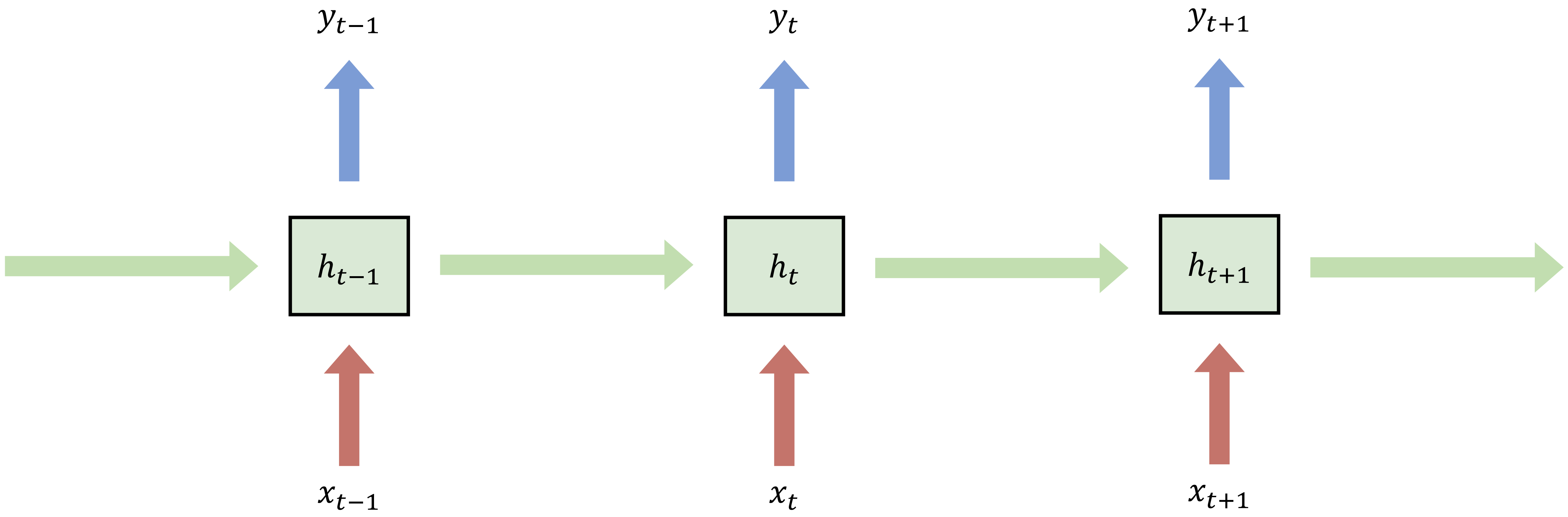
# Training RNN

- RNNs forward through entire sequence to compute loss at each time step's output, then backward through entire sequence to compute gradient
- When training RNN, backpropagation through time is used
  - Passes the error gradient with respect to the weights at each time step into every previous time step



# Backpropagation through time (BPTT)

- BPTT computes derivative of the error w.r.t the weights at every time step

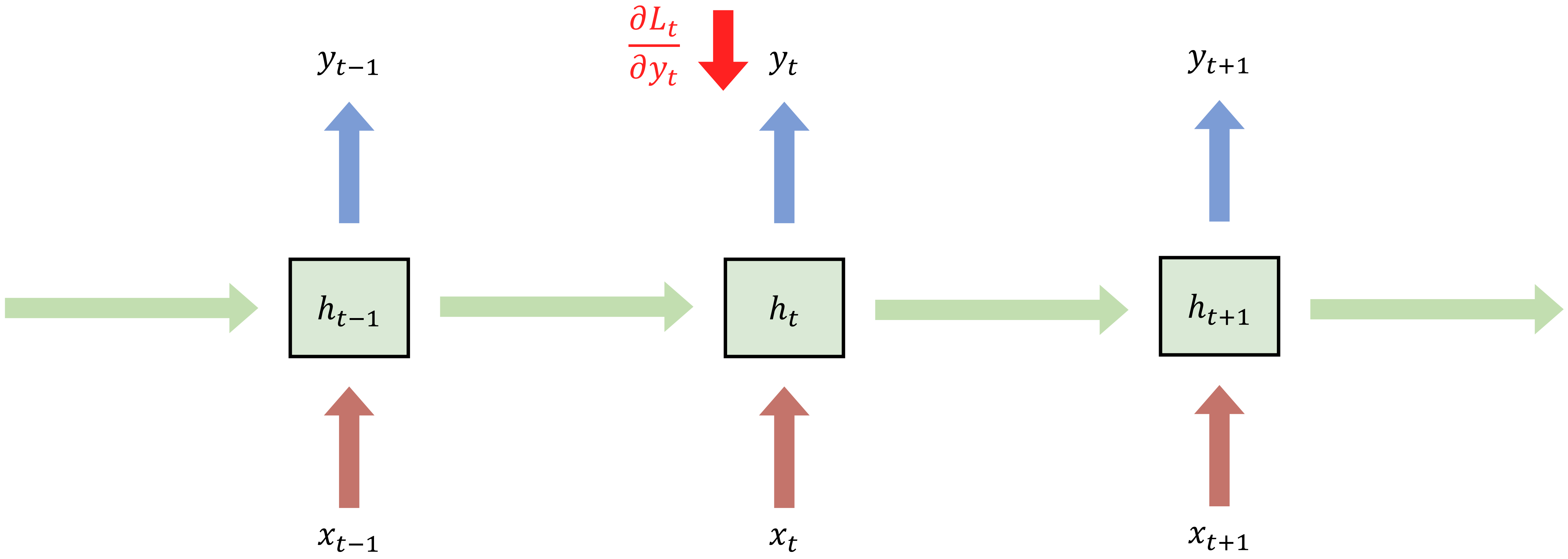



Total loss $L = \sum_{t=0}^{T} L_t$

$$\frac{\partial L_t}{\partial W_{hh}}$$

# Backpropagation through time (BPTT)

- BPTT computes derivative of the error w.r.t the weights at every time step

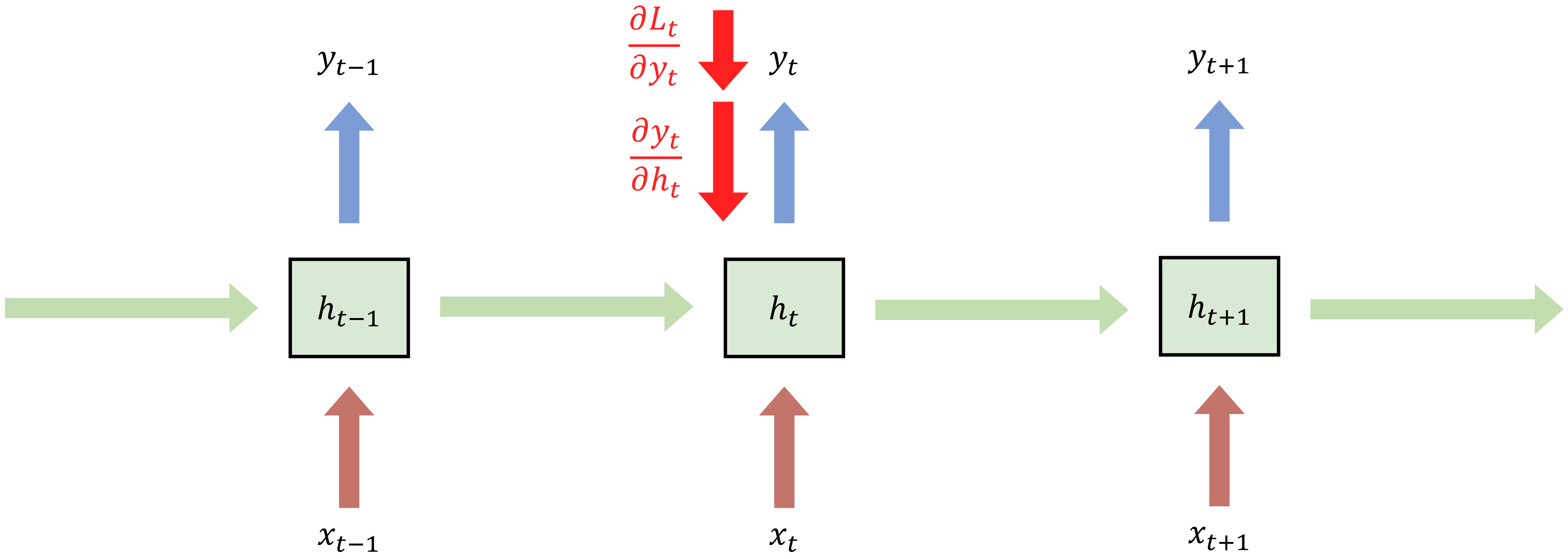



Total loss $L = \sum_{t=0}^{T} L_t$

$$\frac{\partial L_t}{\partial W_{hh}}$$

# Backpropagation through time (BPTT)

- BPTT computes derivative of the error w.r.t the weights at every time step

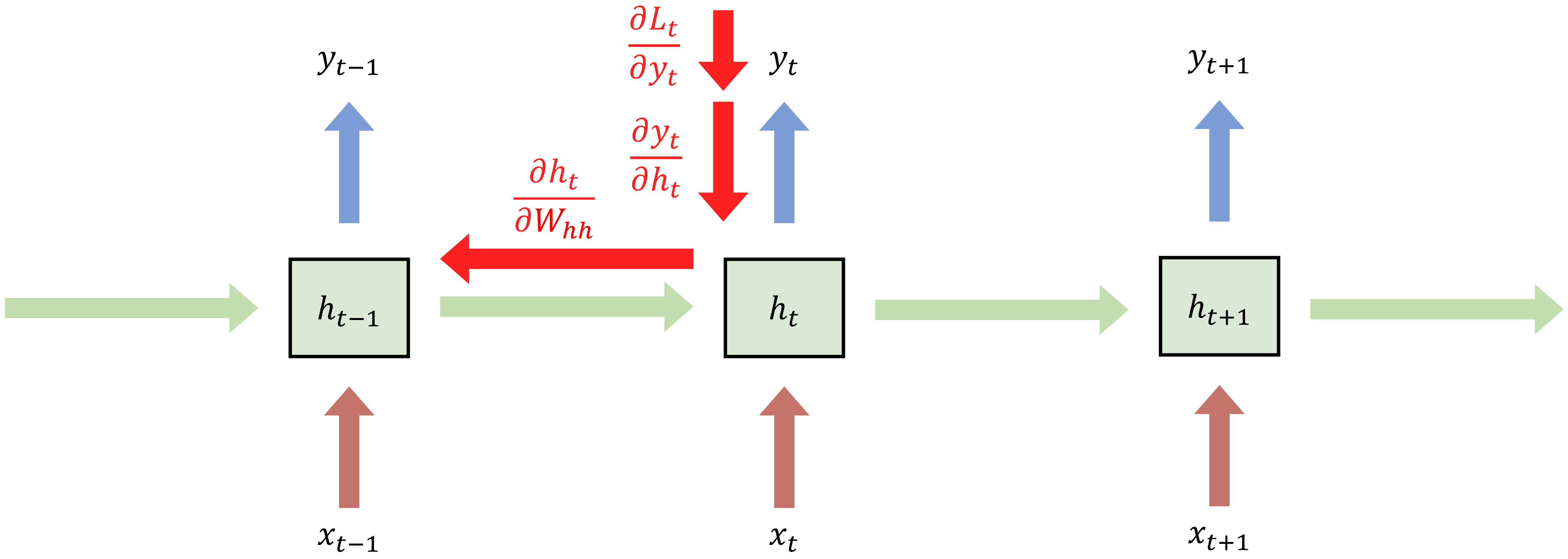



Total loss $L = \sum_{t=0}^{T} L_t$

$$\frac{\partial L_t}{\partial W_{hh}}$$

# Backpropagation through time (BPTT)

- BPTT computes derivative of the error w.r.t the weights at every time step

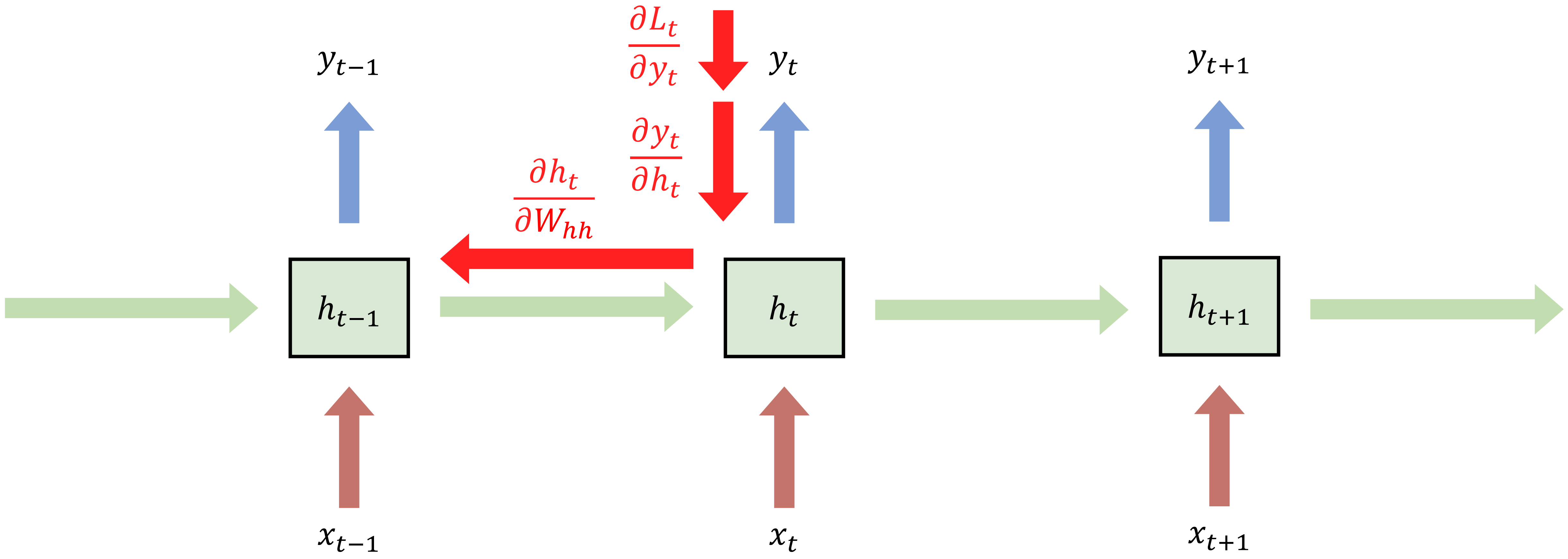



Total loss $L = \sum_{t=0}^{T} L_t$

$$\frac{\partial L_t}{\partial W_{hh}}$$

# Backpropagation through time (BPTT)

- BPTT computes derivative of the error w.r.t the weights at every time step

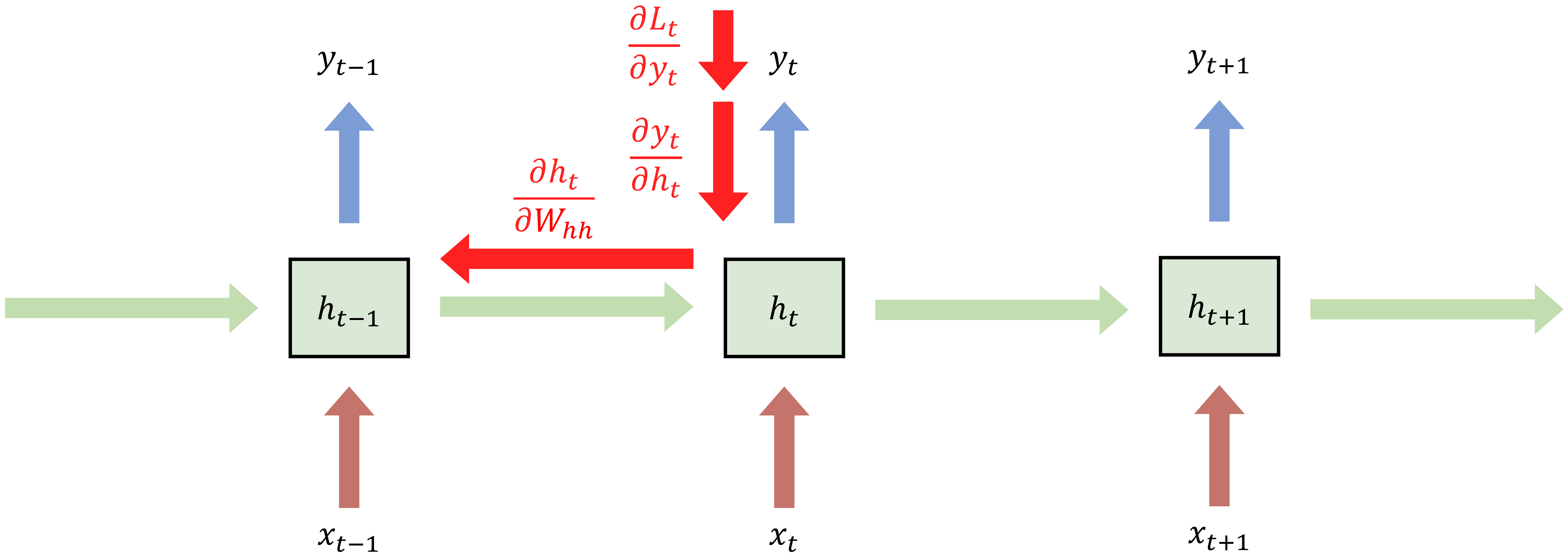



Total loss $L = \sum_{t=0}^{T} L_t$

$$\frac{\partial L_t}{\partial W_{hh}} = \frac{\partial L_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial W_{hh}}$$

# Backpropagation through time (BPTT)

- BPTT computes derivative of the error w.r.t the weights at every time step

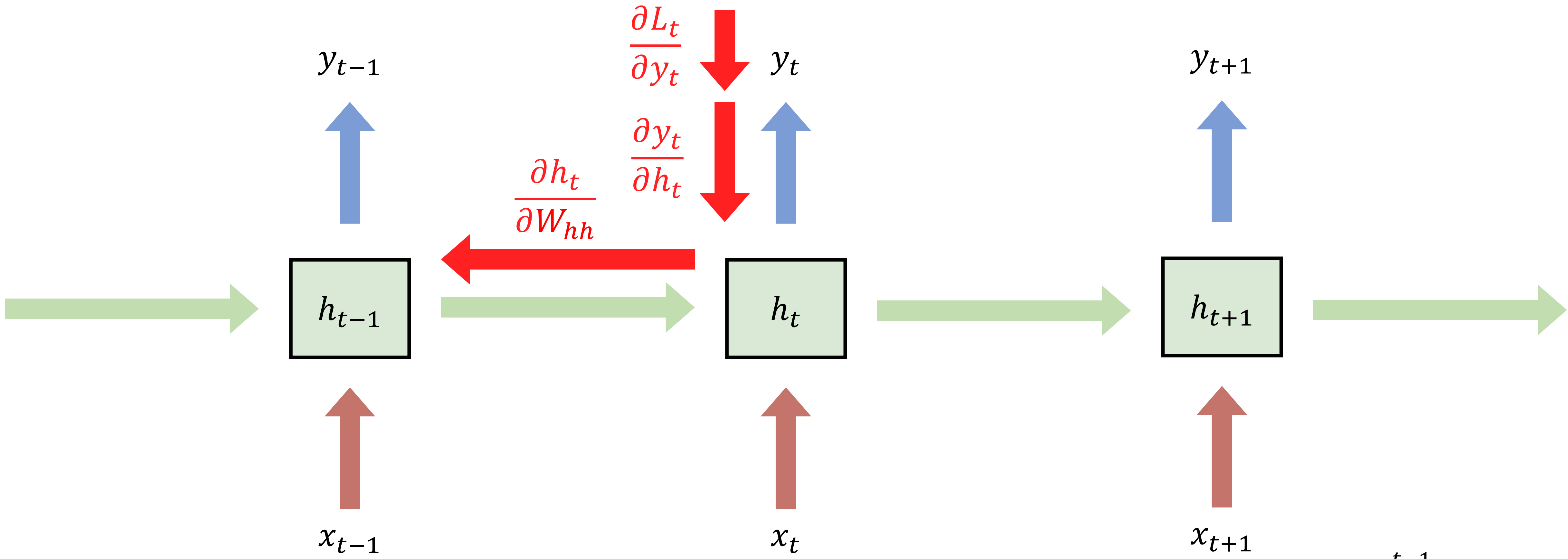



Total loss $L = \sum_{t=0}^{T} L_t$

$$\frac{\partial L_t}{\partial W_{hh}} = \frac{\partial L_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial W_{hh}} = \frac{\partial L_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial h_t} \cdot \sum_{k=0}^{t} \frac{\partial h_t}{\partial h_k} \cdot \frac{\partial h_k}{\partial W_{hh}}$$

# Backpropagation through time (BPTT)

- BPTT computes derivative of the error w.r.t the weights at every time step




Total loss $L = \sum_{t=0}^{T} L_t$

$$\frac{\partial L_t}{\partial W_{hh}} = \frac{\partial L_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial W_{hh}} = \frac{\partial L_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial h_t} \cdot \sum_{k=0}^{t} \frac{\partial h_t}{\partial h_k} \cdot \frac{\partial h_k}{\partial W_{hh}}$$

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k}^{t-1} \frac{\partial h_{j+1}}{\partial h_j}$$

# Vanishing Gradient Problem

- The hidden-to-hidden connections in standard RNN can cause gradient vanishing during backprop
- Vanishing gradient is the problem that the gradient value gets small as the time step accumulates.
- This happens because **small gradient values** (e.g. between 0 and 1 with sigmoid) are multiplied repeatedly

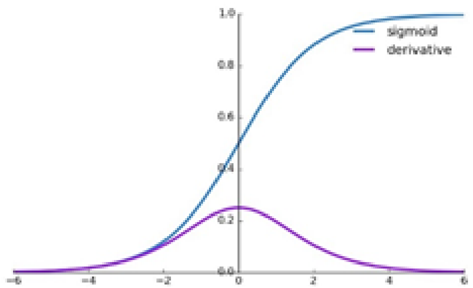
$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k}^{t-1} \frac{\partial h_{j+1}}{\partial h_j}$$

*The girl is sitting on the chair and, … , drinking juice with ______ mother.*

# Vanishing Gradient Problem

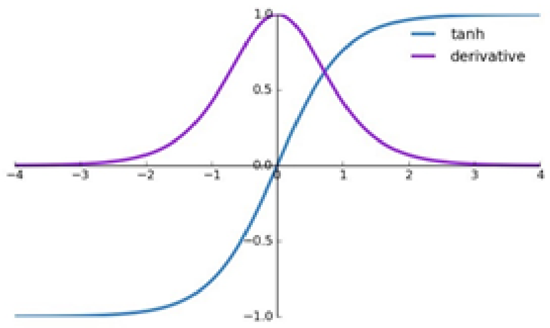
- tanh can prevent vanishing gradient problem to some extent since its gradient value is usually larger than sigmoid's gradient value

❑ Sigmoid




| *Sigmoid* | |
|---|---|
| $f(x)$ | $\frac{1}{1+e^{-x}}$ $(y: 0 \sim 1)$ |
| $\frac{d}{dx}f(x)$ | $\frac{1}{1+e^{-x}}\left(1-\frac{1}{1+e^{-x}}\right)$ $(y': 0 \sim 0.25)$ |

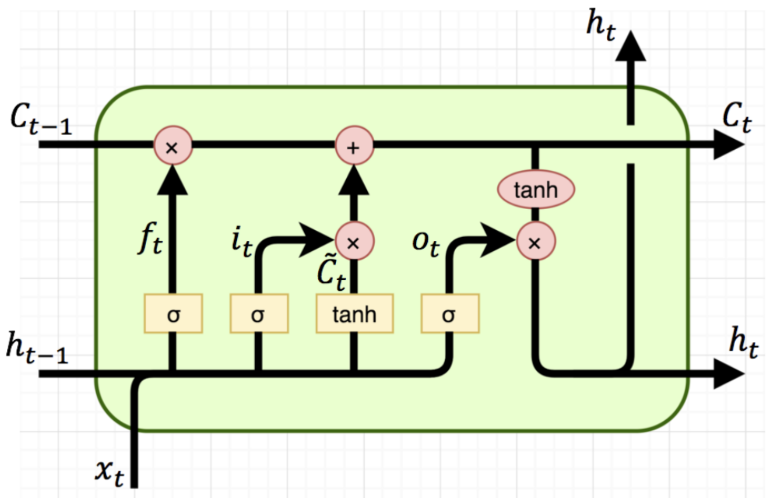
❑ Tanh




| *Tanh* | |
|---|---|
| $f(x)$ | $\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$ $(y: -1 \sim 1)$ |
| $\frac{d}{dx}f(x)$ | $1-\tanh(x)^2$ $(y': 0 \sim 1)$ |

# Long Short-Term Memory (LSTM)

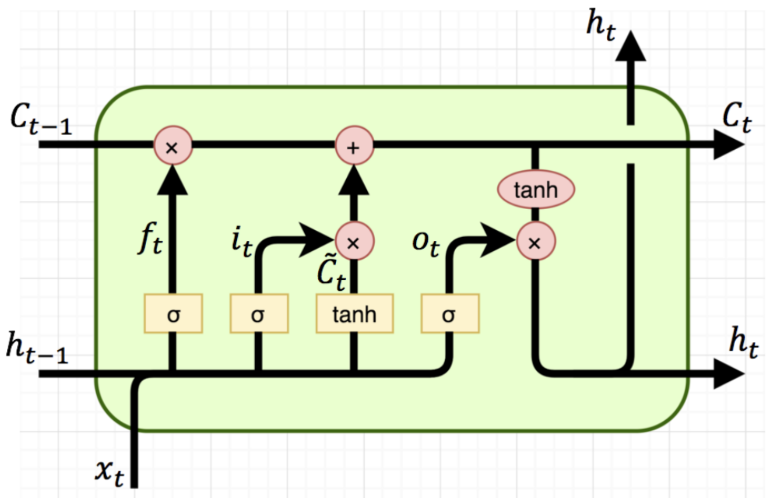
- RNNs addressing vanishing/exploding gradients
  - A recurrent neural network variety designed to retain long-term dependencies
  - Helps dealing with both the vanishing and exploding gradient problem
  - The key idea is an additive connection of previous memories passed through time




(a) Long Short-Term Memory

# Long Short-Term Memory (LSTM)

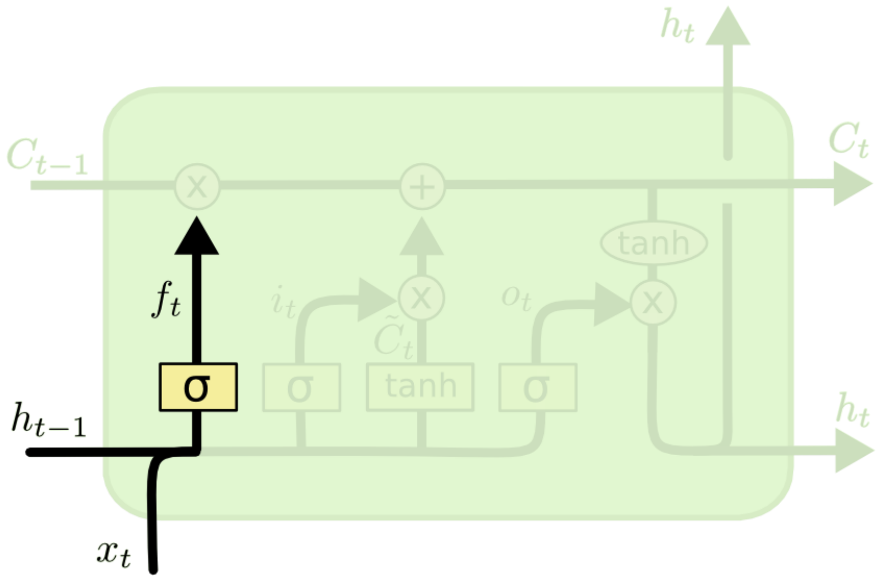
- Overview
  - Information is passed through two variables
  - There are four switch variables that determines how the information flows through time




(a) Long Short-Term Memory

# Long Short-Term Memory (LSTM)

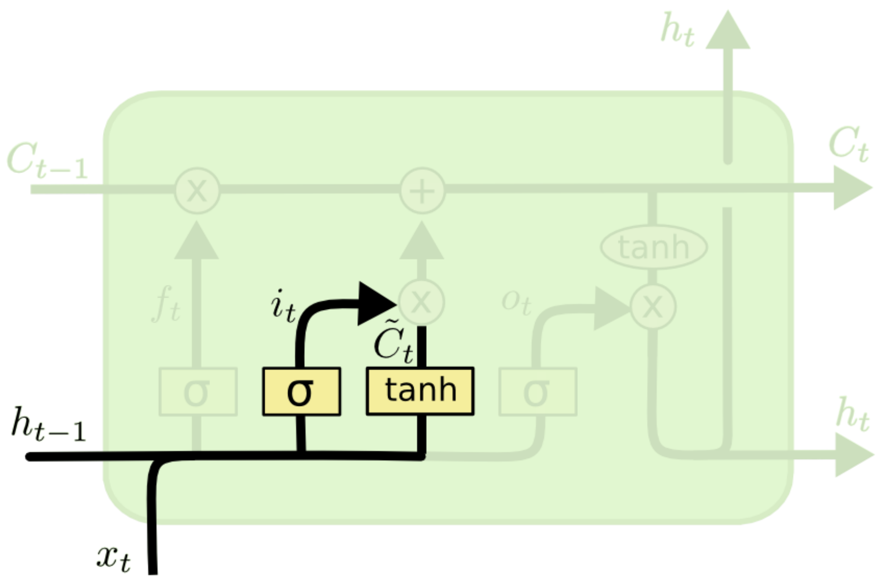
- The forget gate decides whether to throw away information from the previous cell state
  - If $C_{t-1}$ should be removed, small $f_t$ is multiplied to $C_{t-1}$
  - If $C_{t-1}$ should be retained, large $f_t$ is multiplied to $C_{t-1}$




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

# Long Short-Term Memory (LSTM)

- The input gate behaves similarly to forget gates with new inputs
- Candidate new information $\tilde{C}_t$ is computed with the current input and previous hidden units
  - If new information should be added, large $i_t$ is multiplied to $\tilde{C}_t$ and added to $C_{t-1}$
  - If new information should not be added, small $i_t$ is multiplied to $\tilde{C}_t$ and added to $C_{t-1}$

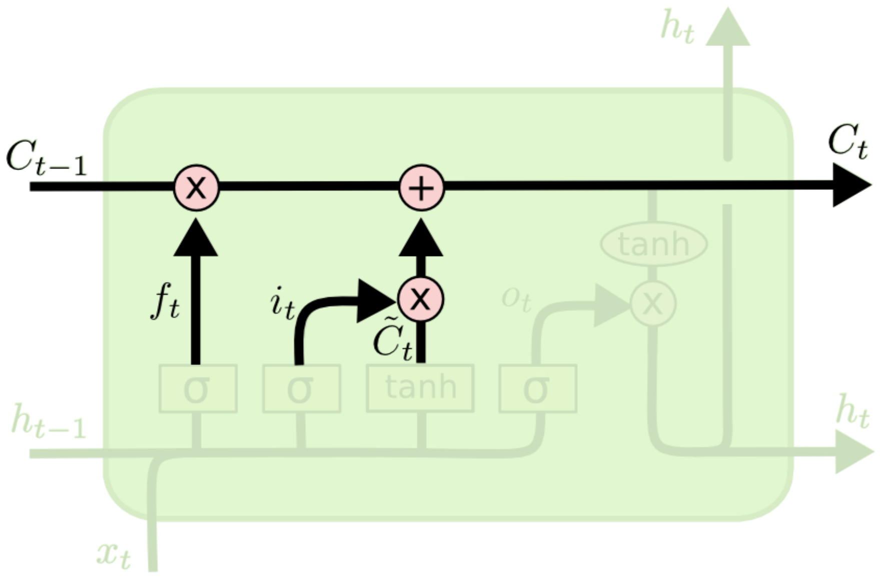



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$

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

# Long Short-Term Memory (LSTM)

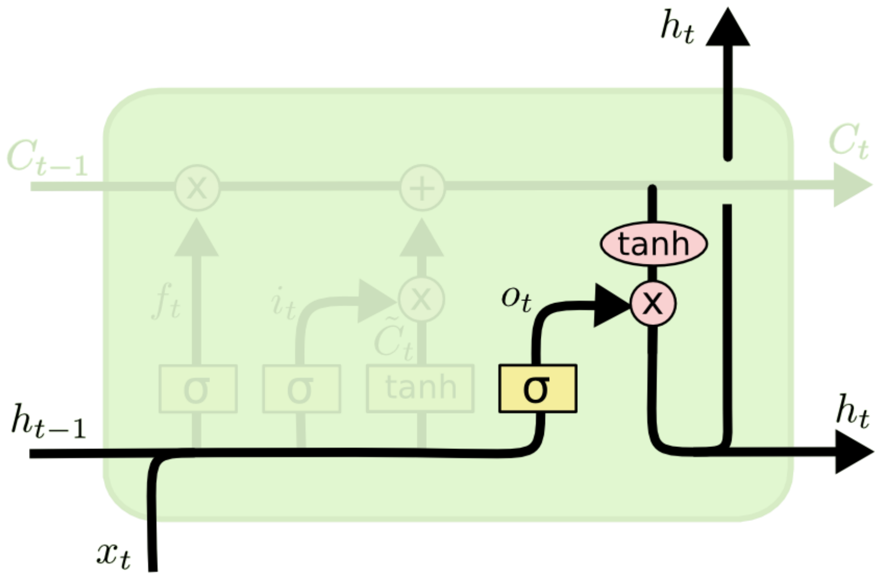
- The new cell state is calculated as weighted summing between the previous cell state and the newly created candidate information




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

# Long Short-Term Memory (LSTM)

- An output gate determines what to output from the current cell state for the next hidden state in the LSTM




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

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

# Preprocessing text data

- To handle the text dataset, we first need to transform the data into a form that can be used for training
- Two common methods for preprocessing text data

  (1) One-hot encoding

  (2) Word embedding

# Revisit : One-hot Encoding

- One-hot encoding is a very sparse vector representation of sentences
- Representing each word in the vocabulary as "one-hot" encoded index
- To represent each word
  (1) create a zero vector with length equal to the number of the vocabularies
  (2) place one in the index that corresponds to the word

**One-hot encoding**

| | | cat | mat | on | sat | the |
|---|---|---|---|---|---|---|
| **the** | => | 0 | 0 | 0 | 0 | 1 |
| **cat** | => | 1 | 0 | 0 | 0 | 0 |
| **sat** | => | 0 | 0 | 0 | 1 | 0 |
| ... | | ... | | | | |

Example sentence :
"The cat sat on the mat"

# Revisit : One-hot Encoding

- One-hot encoding has some downsides

  (1) One-hot encoding is very inefficient

  - An one-hot encoded vector is sparse (meaning most indicies are zero)

  (2) One-hot encoding cannot incorporate semantics between each word

  - Every token in one-hot encoding is equally distant away from all the others

# Revisit : One-hot Encoding

- One-hot encoding has some downsides

  (1) One-hot encoding is very inefficient

    - An one-hot encoded vector is sparse (meaning most indicies are zero)

  (2) One-hot encoding cannot incorporate semantics between each word

    - Every token in one-hot encoding is equally distant away from all the others
    - It would be good if similar words are encoded into similar vectors

# Word embeddings

- Word embeddings give us a way to use an efficient, dense representation in which similar words have a similar encoding
- Importantly, we do not have to specify this encoding by hand
  - they are trainable parameters : weights are learned by the model during training
  - the neural network captures the token's meaning as a vector
- Word embeddings can be 8-dimensional up to 1024-dimensions
- A higher dimensional embedding can capture fine-grained relationships between words, but takes more data and time to learn

**A 4-dimensional embedding**

| | | | | | |
|---|---|---|---|---|---|
| **cat** | => | 1.2 | -0.1 | 4.3 | 3.2 |
| **mat** | => | 0.4 | 2.5 | -0.9 | 0.5 |
| **on** | => | 2.1 | 0.3 | 0.1 | 0.4 |
| … | | … | | | |

# Example of word embeddings

- Nearest words for query word ‘study’ :

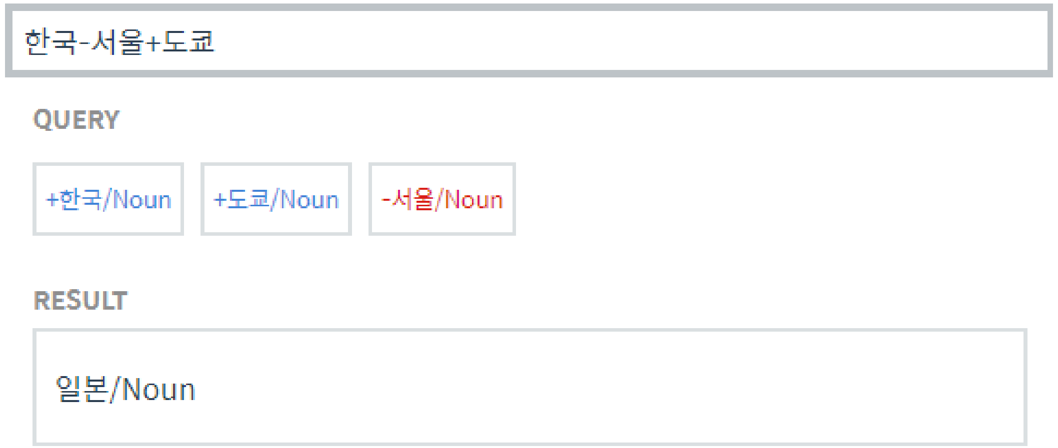
  - Research, analysis, discovery

- Nearest words for query word ‘skill’ :

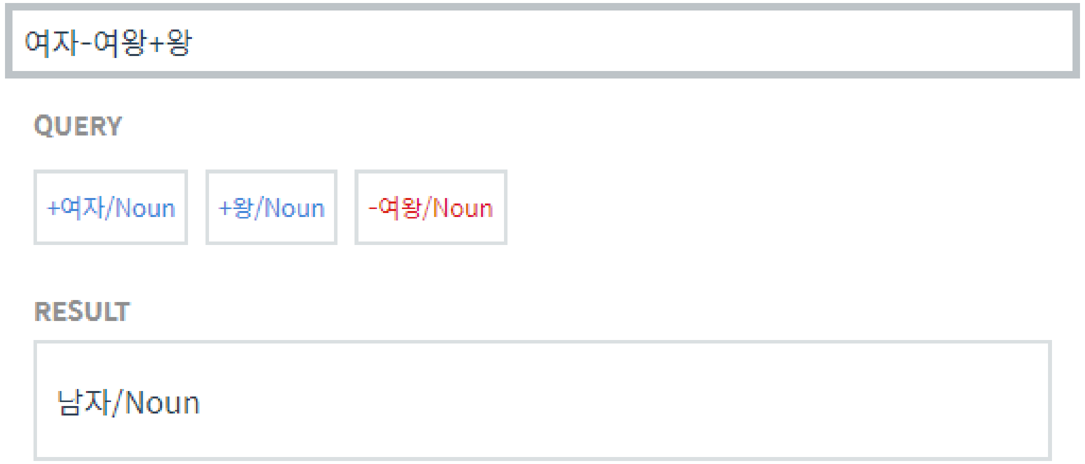
  - ability, strength, talent

# Example of word embeddings

- We can consider embedded words have relations




Korea – Seoul + Tokyo = Japan




woman – queen + man = king

# Papers related to word-embeddings

- CBoW, Skip-gram: https://arxiv.org/pdf/1301.3781v3.pdf
- GloVe: https://www.aclweb.org/anthology/D14-1162.pdf
- fastText: https://arxiv.org/pdf/1607.04606v2.pdf