Lecture 14

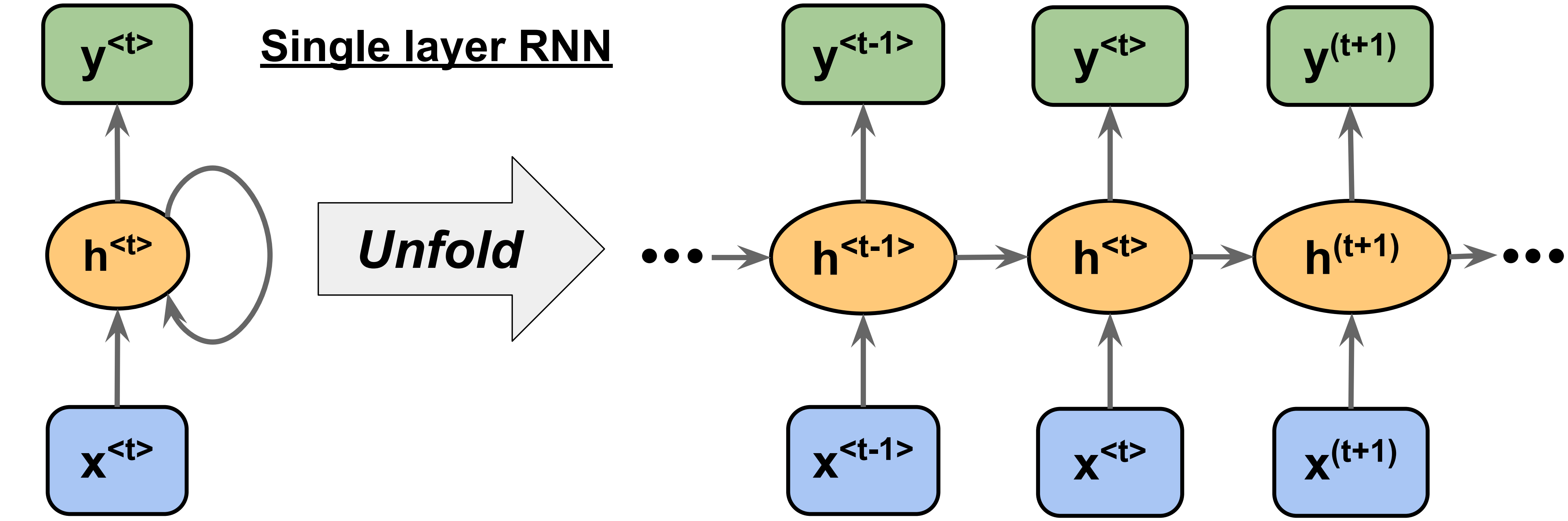
Introduction to Recurrent Neural Networks (Part 2)

STAT 479: Deep Learning, Spring 2019

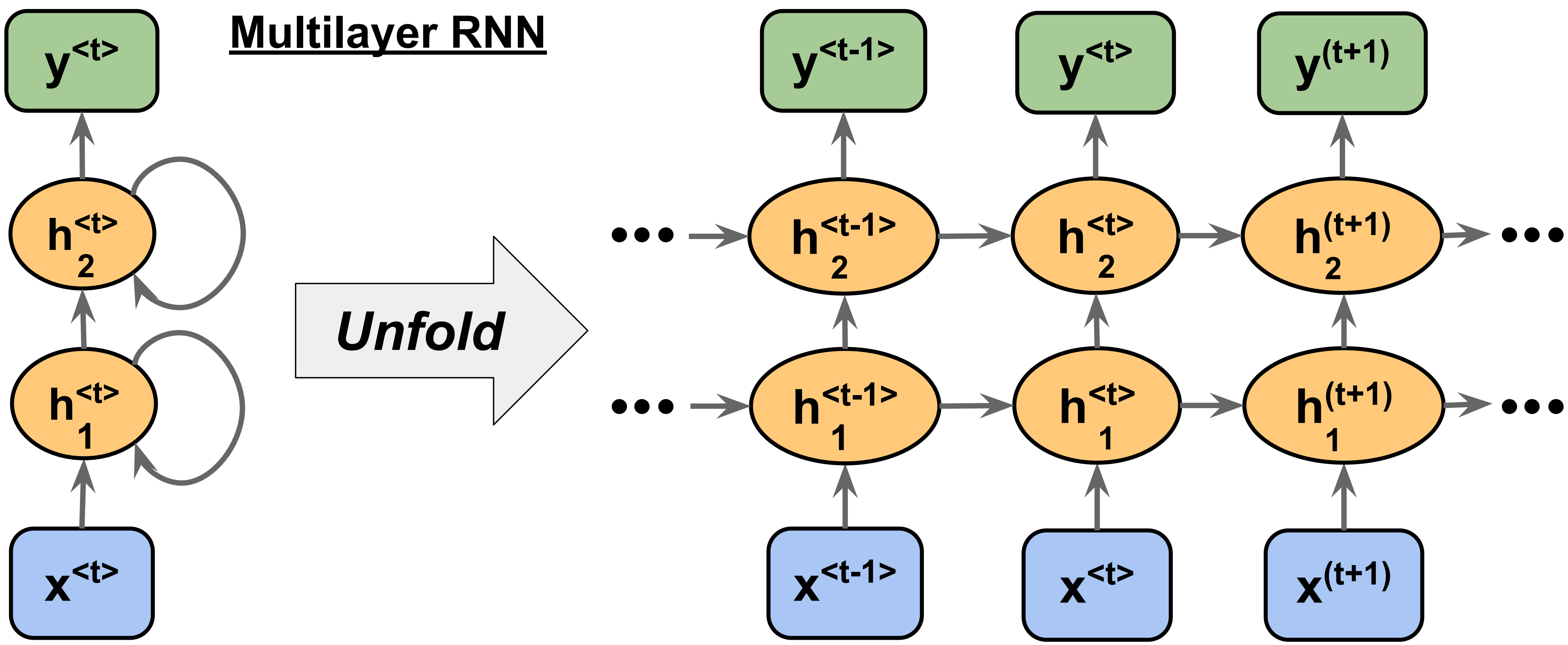
Sebastian Raschka

http://stat.wisc.edu/~sraschka/teaching/stat479-ss2019/

Single layer RNN

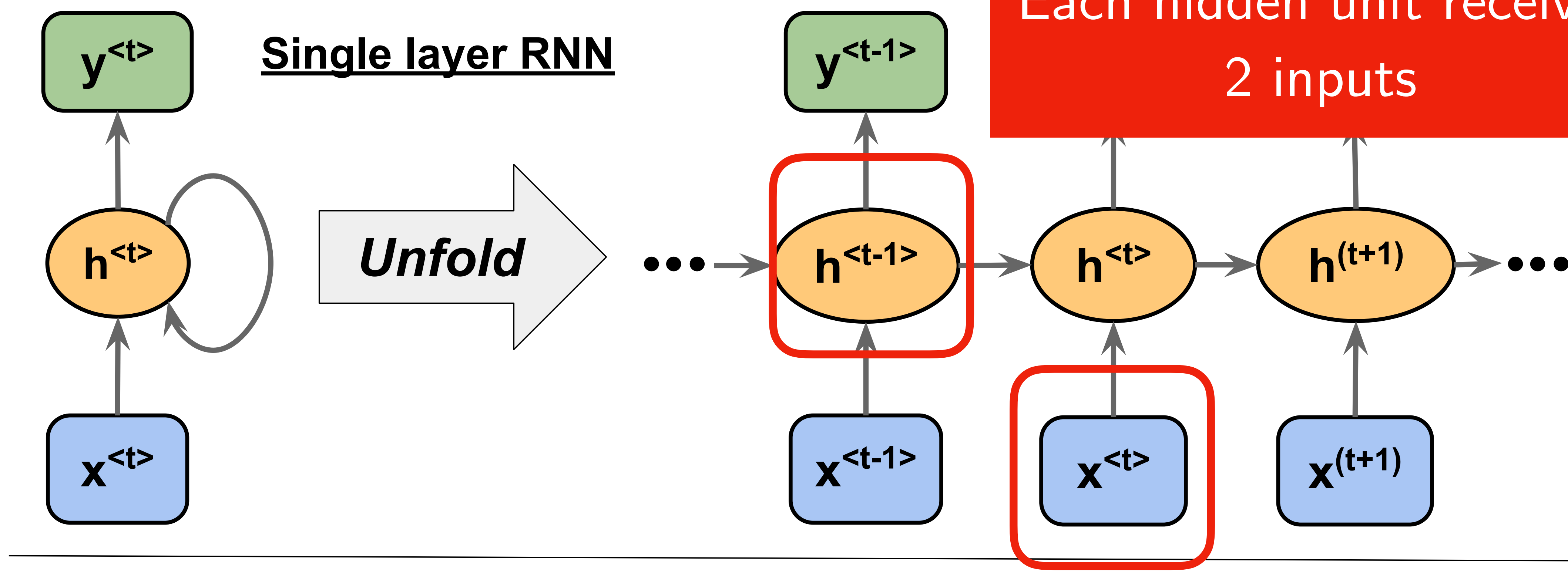


Multilayer RNN

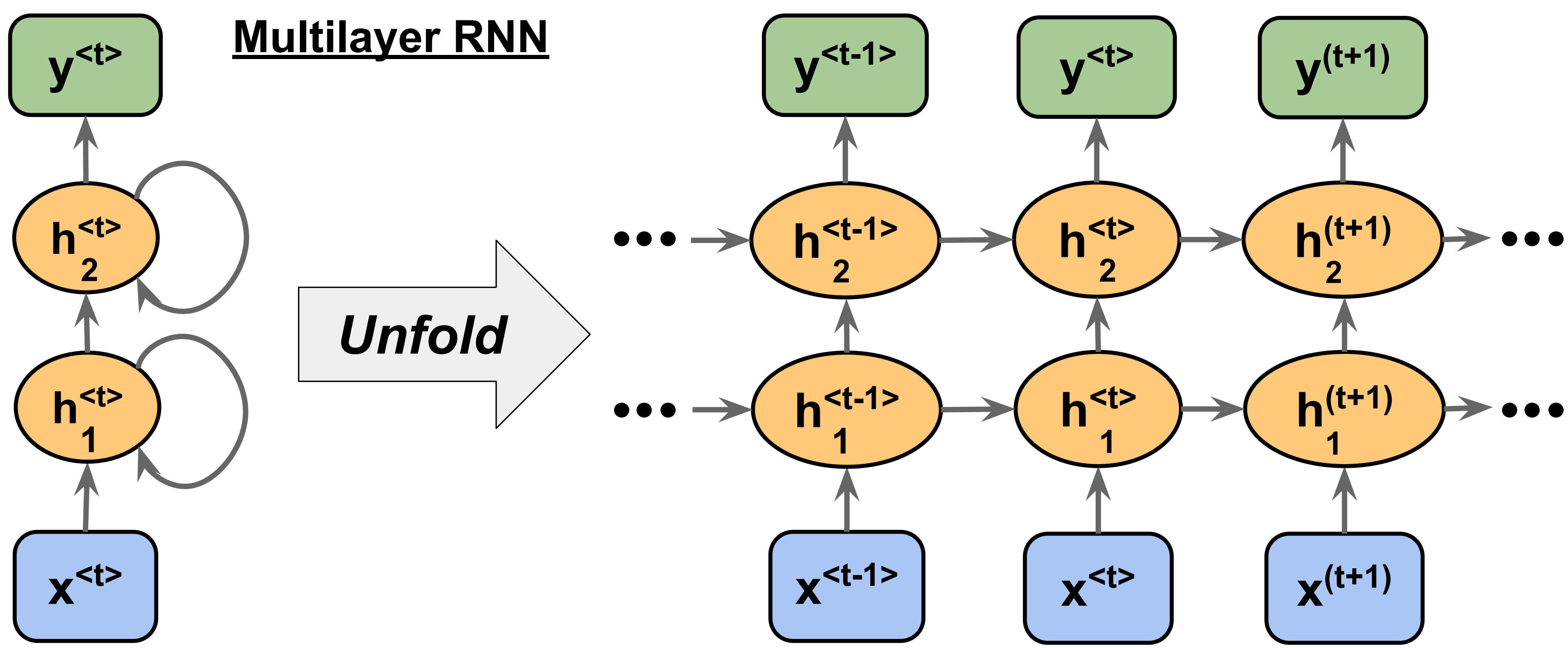


Each hidden unit receives 2 inputs

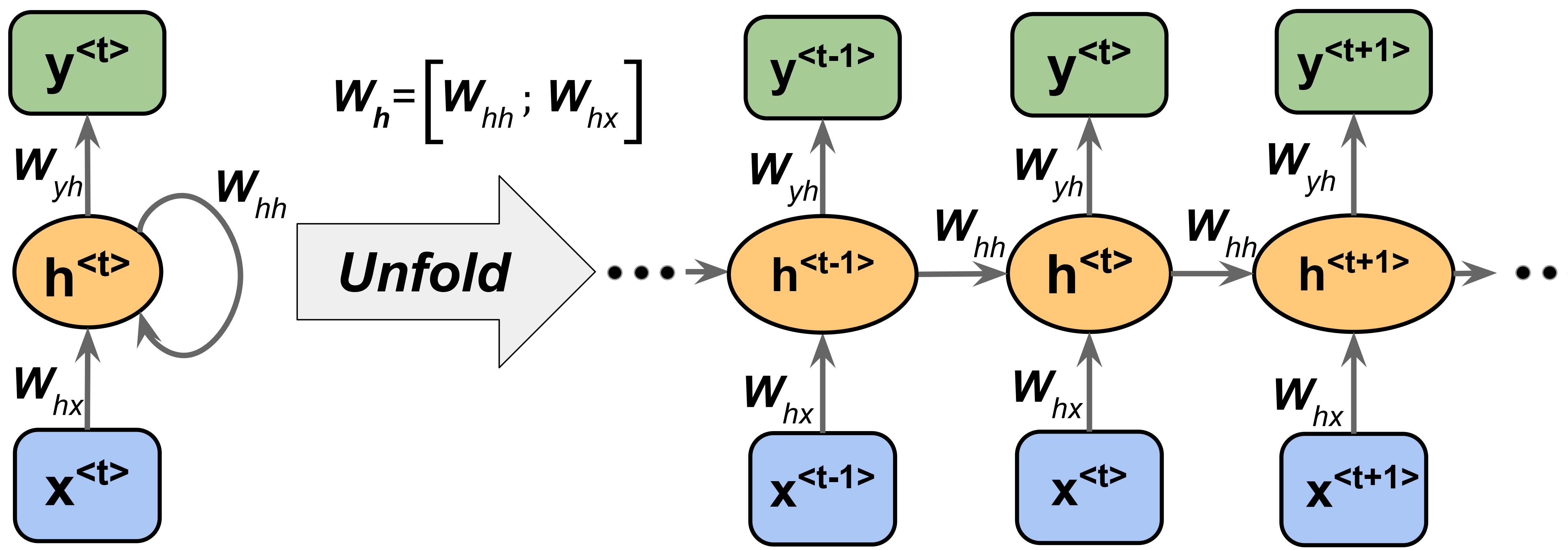
Single layer RNN



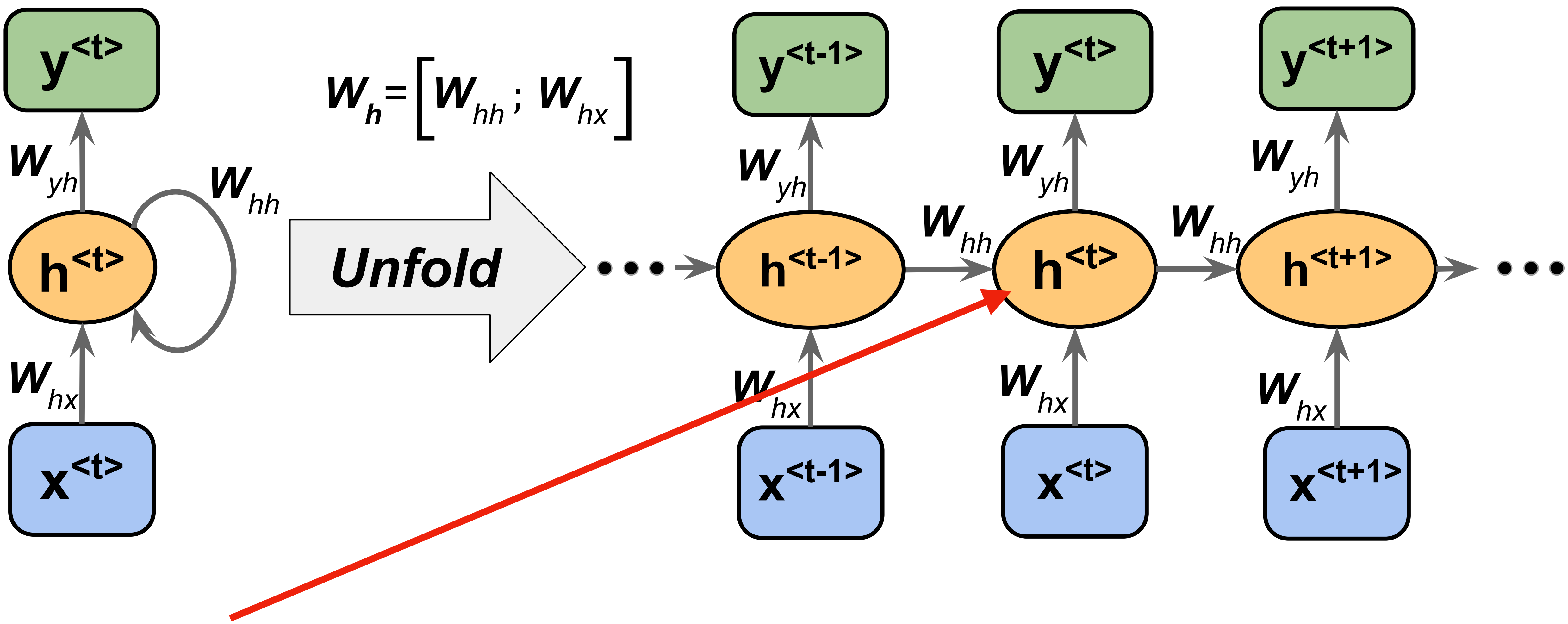
Multilayer RNN



Weight matrices in a single-hidden layer RNN



Weight matrices in a single-hidden layer RNN



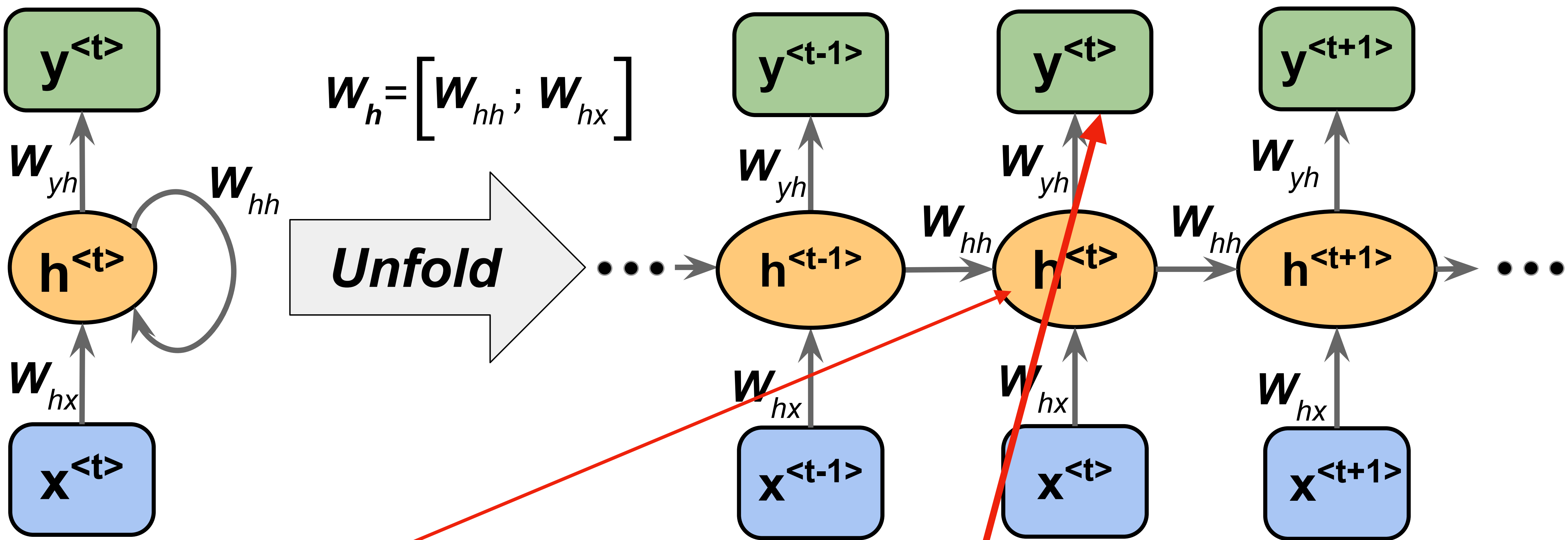
Net input:

$$\mathbf{z}_h^{\langle t \rangle} = \mathbf{W}_{hx}\mathbf{x}^{\langle t \rangle} + \mathbf{W}_{hh}\mathbf{h}^{\langle t-1 \rangle} + \mathbf{b}_h$$

Activation:

$$\mathbf{h}^{\langle t \rangle} = \sigma_h\left(\mathbf{z}_h^{\langle t \rangle}\right)$$

Weight matrices in a single-hidden layer RNN



Net input:

$$\mathbf{z}_h^{\langle t \rangle} = \mathbf{W}_{hx}\mathbf{x}^{\langle t \rangle} + \mathbf{W}_{hh}\mathbf{h}^{\langle t-1 \rangle} + \mathbf{b}_h$$

Activation:

$$\mathbf{h}^{\langle t \rangle} = \sigma_h\left(\mathbf{z}_h^{\langle t \rangle}\right)$$

Net input:

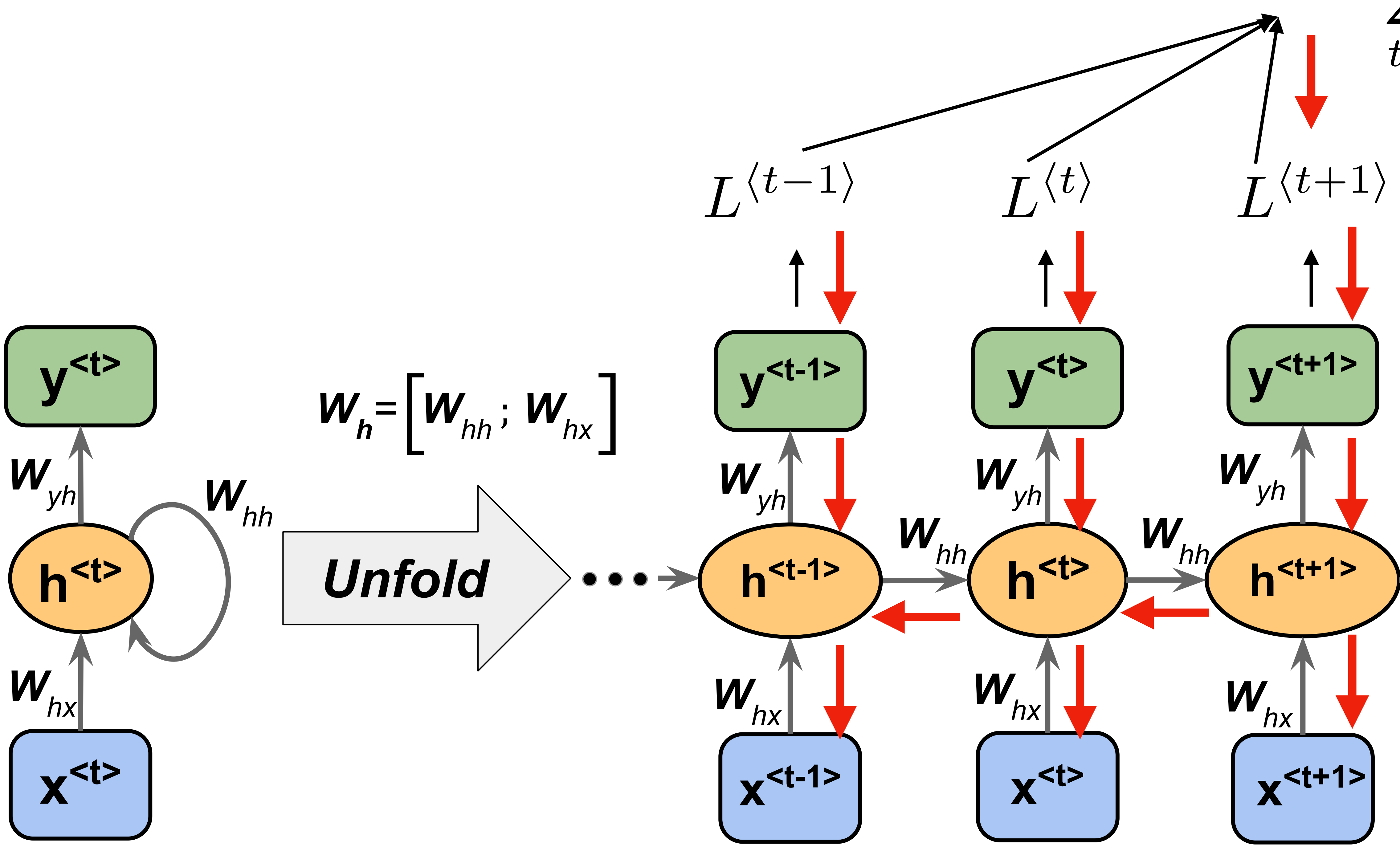
$$\mathbf{z}_y^{\langle t \rangle} = \mathbf{W}_{yh}\mathbf{h}^{\langle t \rangle} + \mathbf{b}_y$$

Output:

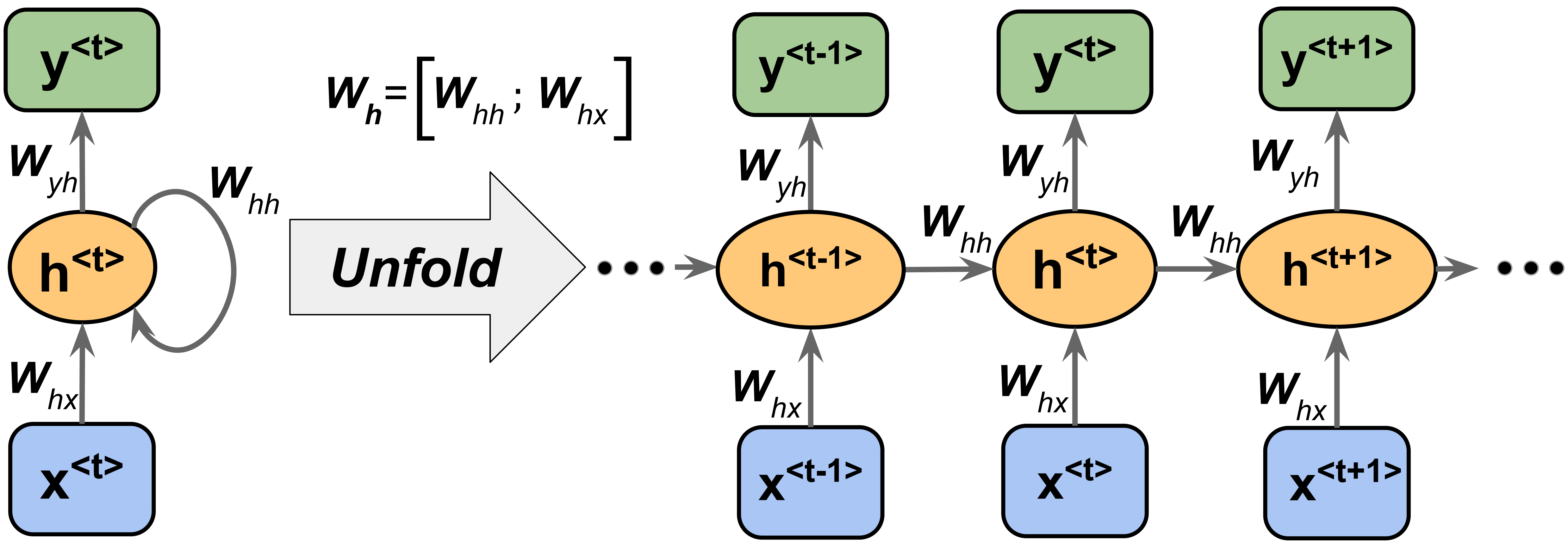
$$\mathbf{y}^{\langle t \rangle} = \sigma_y\left(\mathbf{z}_y^{\langle t \rangle}\right)$$

Backpropagation through time

$$L = \sum_{t=1}^{T} L^{\langle t \rangle}$$



Backpropagation through time

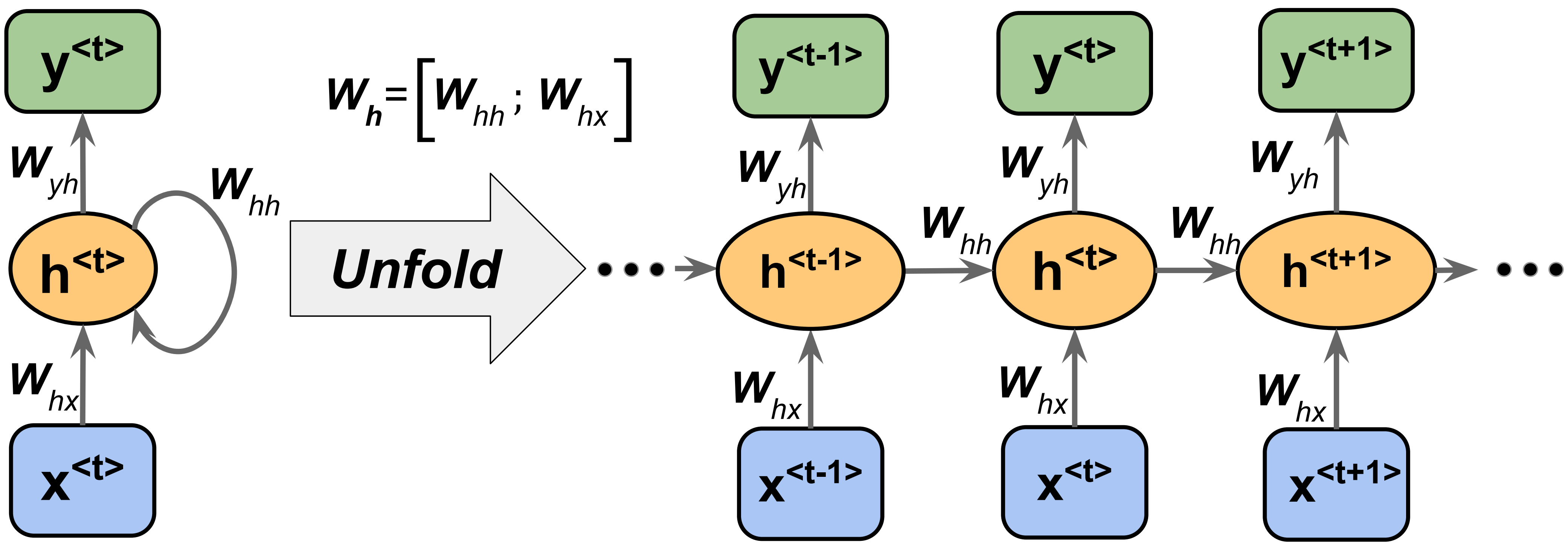


Werbos, Paul J. "Backpropagation through time: what it does and how to do it." *Proceedings of the IEEE* 78, no. 10 (1990): 1550-1560.

The loss is computed as the sum over all time steps:

$$L = \sum_{t=1}^{T} L^{\langle t \rangle}$$

Backpropagation through time

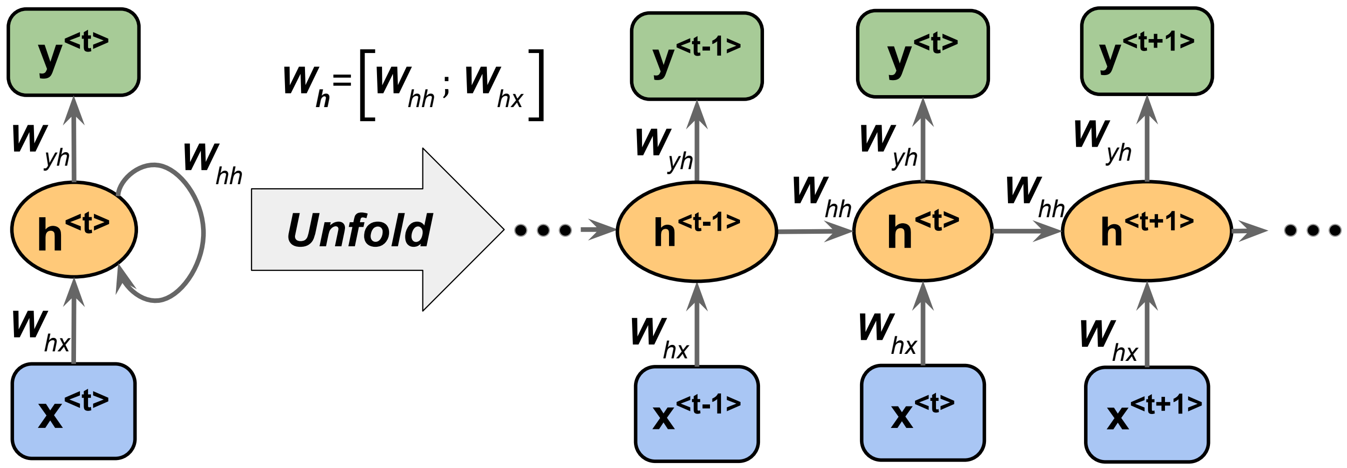


Werbos, Paul J. "[Backpropagation through time: what it does and how to do it.]" *Proceedings of the IEEE* 78, no. 10 (1990): 1550-1560.

$$L = \sum_{t=1}^{T} L^{(t)}$$

$$\frac{\partial L^{(t)}}{\partial \mathbf{W}_{hh}} = \frac{\partial L^{(t)}}{\partial y^{(t)}} \cdot \frac{\partial y^{(t)}}{\partial \mathbf{h}^{(t)}} \cdot \left(\sum_{k=1}^{t} \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} \cdot \frac{\partial \mathbf{h}^{(k)}}{\partial \mathbf{W}_{hh}} \right)$$

Backpropagation through time



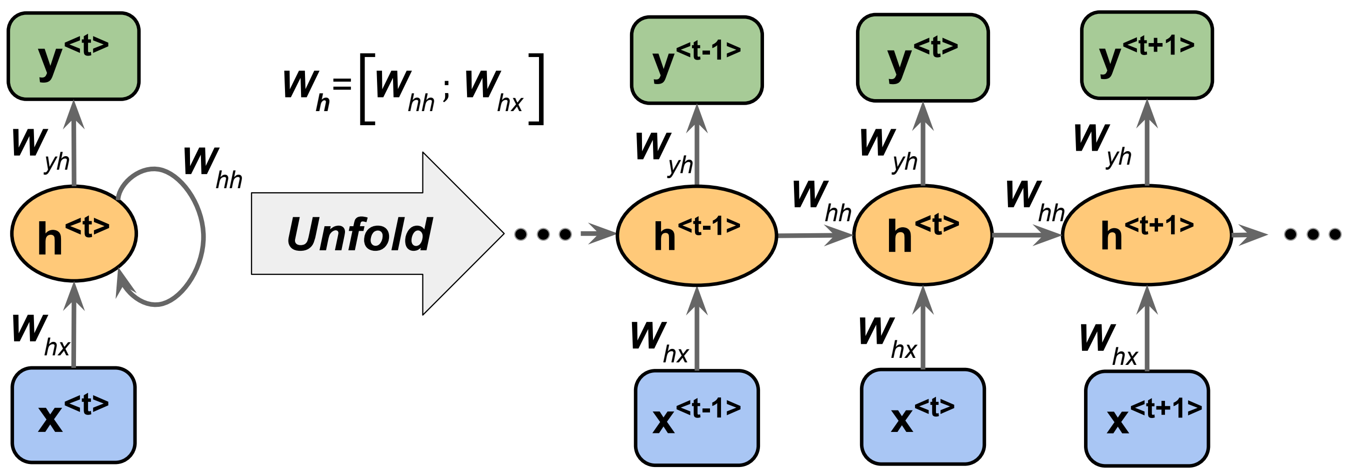
Werbos, Paul J. "Backpropagation through time: what it does and how to do it." *Proceedings of the IEEE* 78, no. 10 (1990): 1550-1560.

$$L = \sum_{t=1}^{T} L^{(t)} \qquad \frac{\partial L^{(t)}}{\partial \mathbf{W}_{hh}} = \frac{\partial L^{(t)}}{\partial y^{(t)}} \cdot \frac{\partial y^{(t)}}{\partial \mathbf{h}^{(t)}} \cdot \left(\sum_{k=1}^{t} \boxed{\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}}} \cdot \frac{\partial \mathbf{h}^{(k)}}{\partial \mathbf{W}_{hh}} \right)$$

computed as a multiplication of adjacent time steps:

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} = \prod_{i=k+1}^{t} \frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}}$$

Backpropagation through time



Werbos, Paul J. "Backpropagation through time: what it does and how to do it." *Proceedings of the IEEE* 78, no. 10 (1990): 1550-1560.

$$L = \sum_{t=1}^{T} L^{(t)} \qquad \frac{\partial L^{(t)}}{\partial \mathbf{W}_{hh}} = \frac{\partial L^{(t)}}{\partial y^{(t)}} \cdot \frac{\partial y^{(t)}}{\partial \mathbf{h}^{(t)}} \cdot \left(\sum_{k=1}^{t} \boxed{\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}}} \cdot \frac{\partial \mathbf{h}^{(k)}}{\partial \mathbf{W}_{hh}} \right)$$

computed as a multiplication of adjacent time steps:

This is very problematic:
Vanishing/Exploding gradient problem!

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} = \prod_{i=k+1}^{t} \frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}}$$

A good resource that explains *backpropation through time* nicely:

Boden, Mikael. "A guide to recurrent neural networks and backpropagation." *the Dallas project* (2002).

Solutions to the vanishing/exploding gradient problems

1) Gradient Clipping: set a max value for gradients if they grow to large (solves only exploding gradient problem)

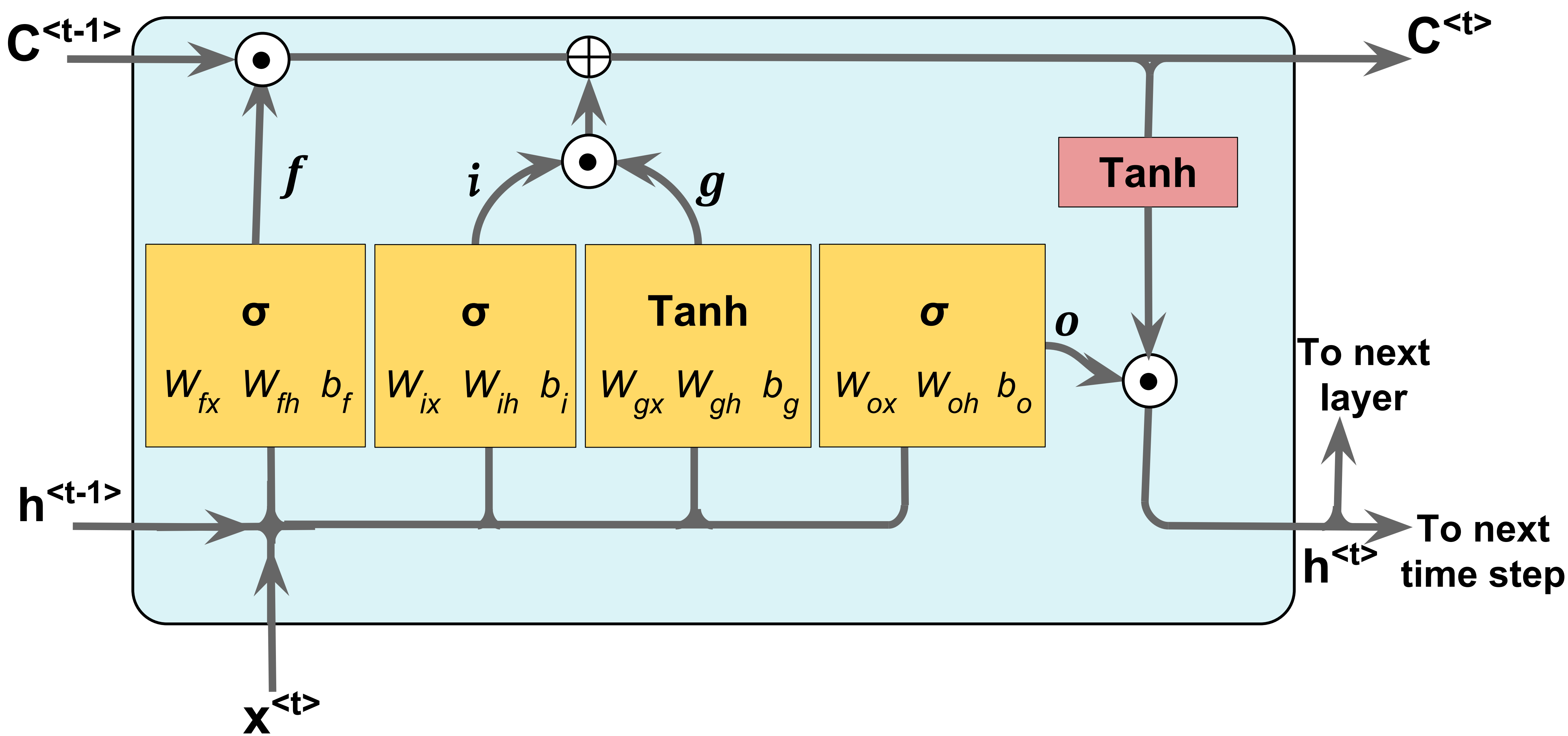
2) Truncated backpropagation through time (TBPTT)
   - simply limits the number of time steps the signal can backpropagate each forward pass. E.g., even if the
   sequence has 100 elements/steps, we may only backpropagate through 20 or so

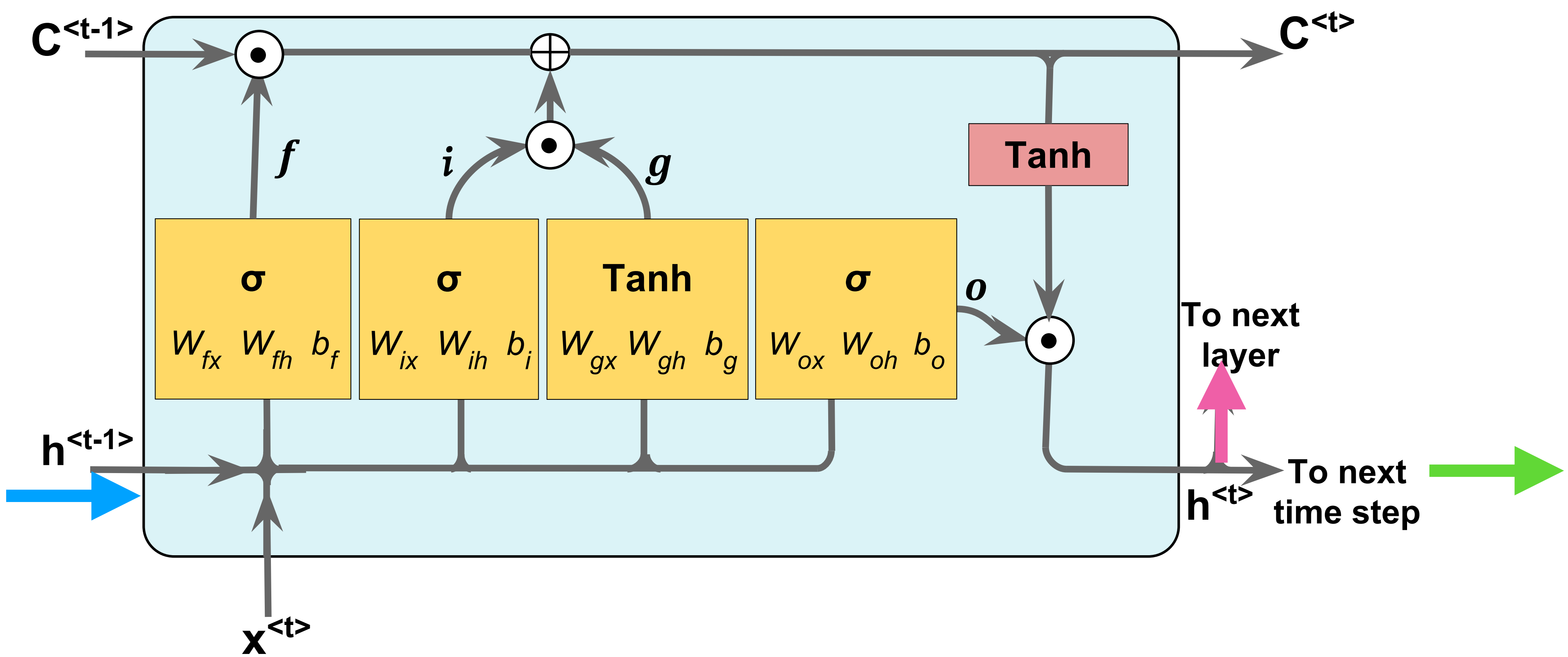
3) Long short-term memory (LSTM) -- uses a memory cell for modeling long-range dependencies and avoid vanishing gradient problems

Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory."
Neural computation 9, no. 8 (1997): 1735-1780.

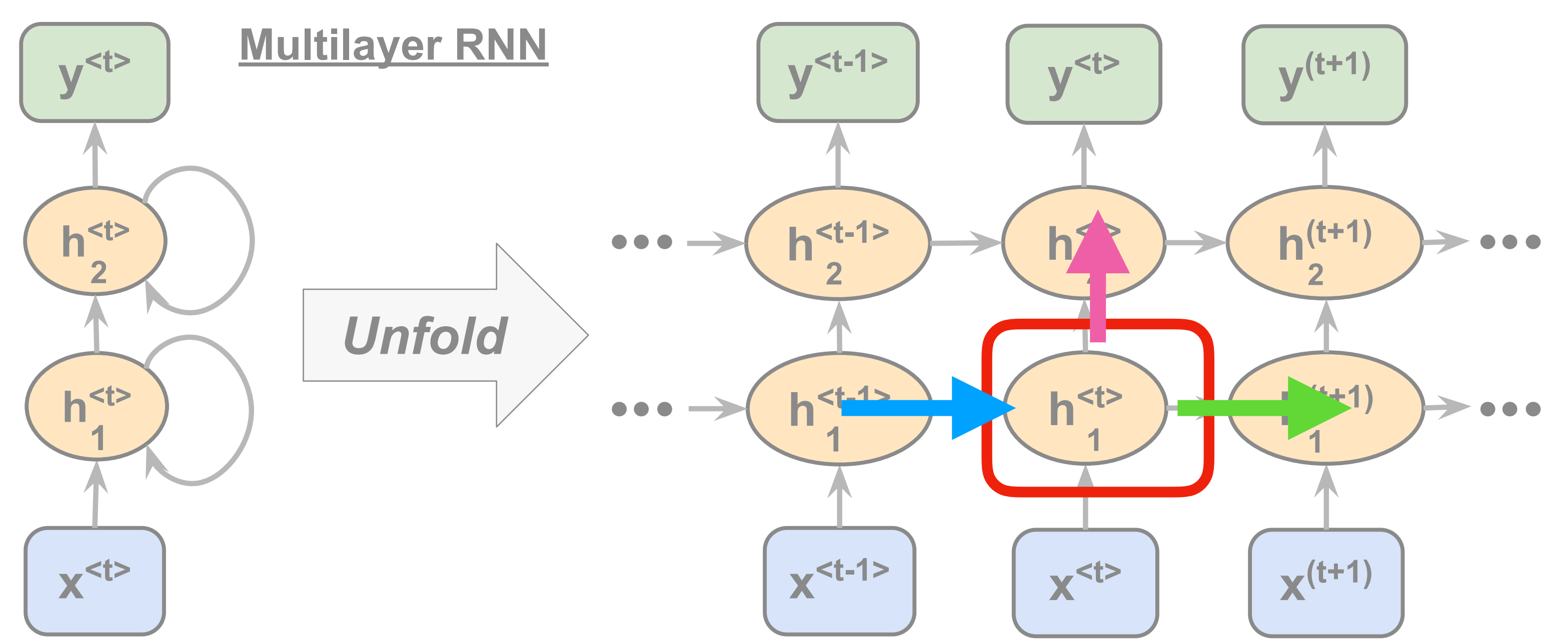
Long-short term memory (LSTM)

LSTM cell:





C<t-1>
C<t>
f
i
g
o
Tanh
σ
Wfx Wfh bf
σ
Wix Wih bi
Tanh
Wgx Wgh bg
σ
Wox Woh bo
To next layer
h<t-1>
h<t>
To next time step
x<t>

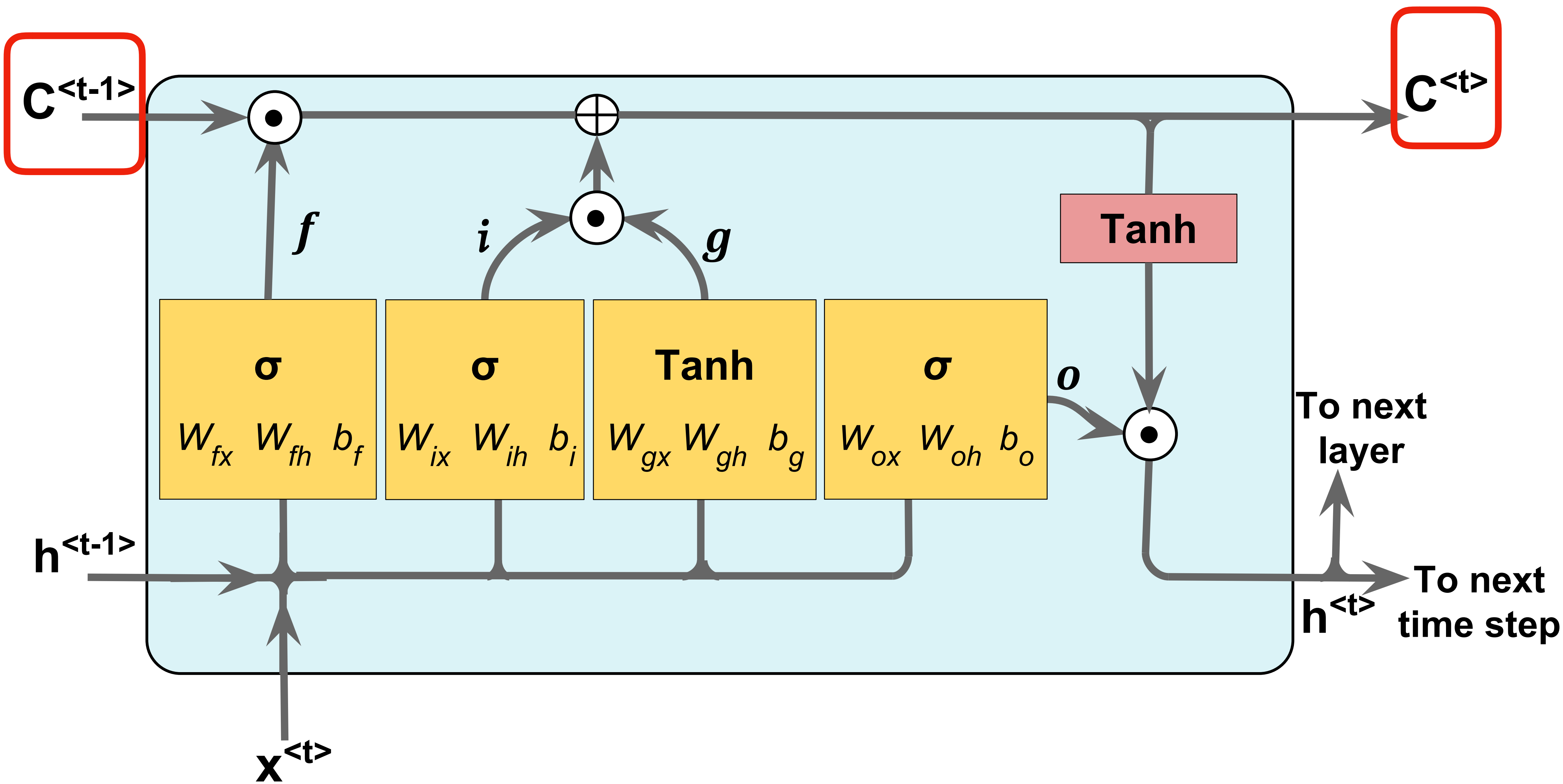


Multilayer RNN
Unfold
y<t>
h<t> 2
h<t> 1
x<t>
y<t-1>
y<t>
y(t+1)
h<t-1> 2
h(t+1) 2
h<t> 1
x<t-1>
x<t>
x(t+1)

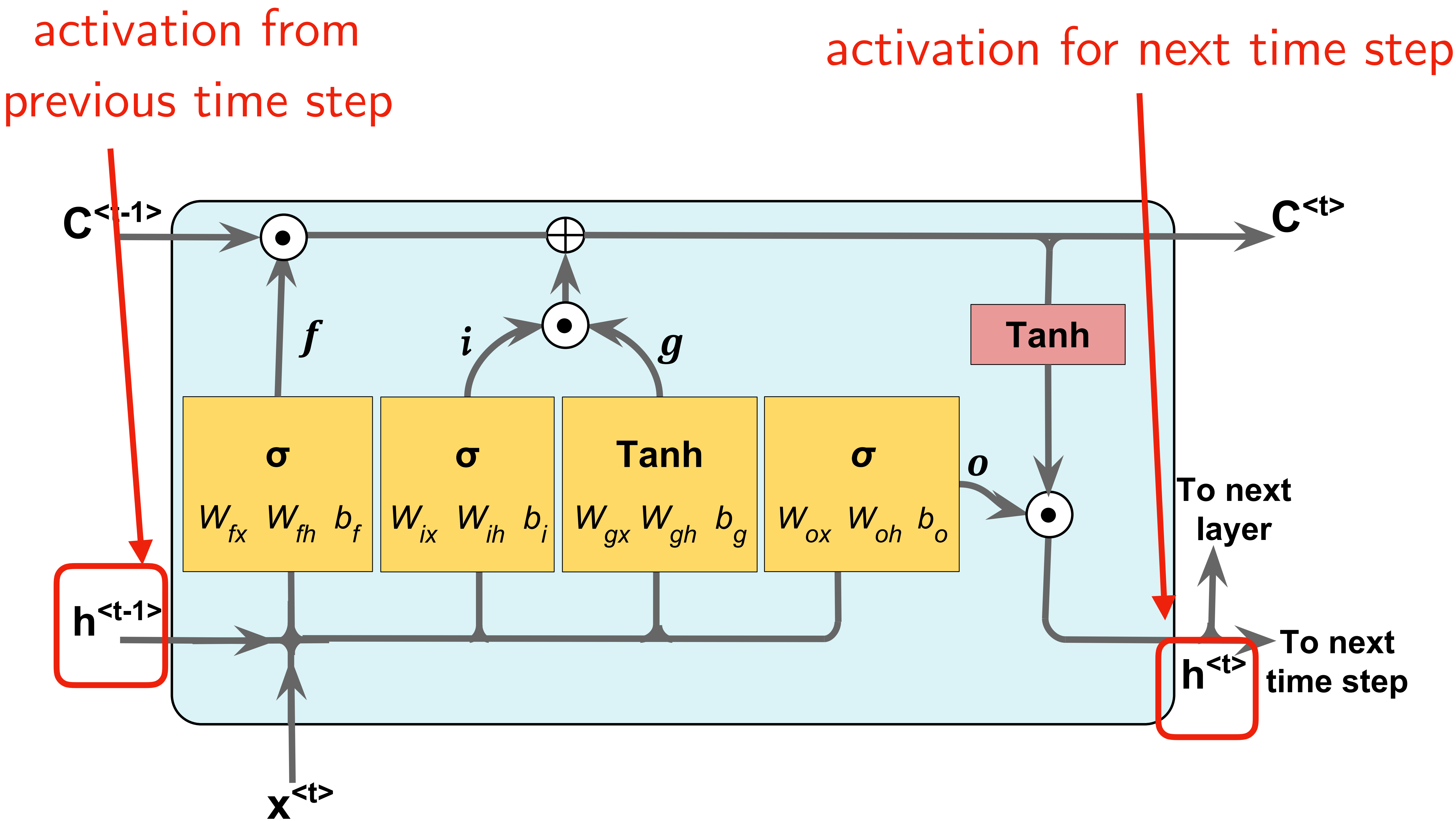
Long-short term memory (LSTM)

Cell state at previous time step

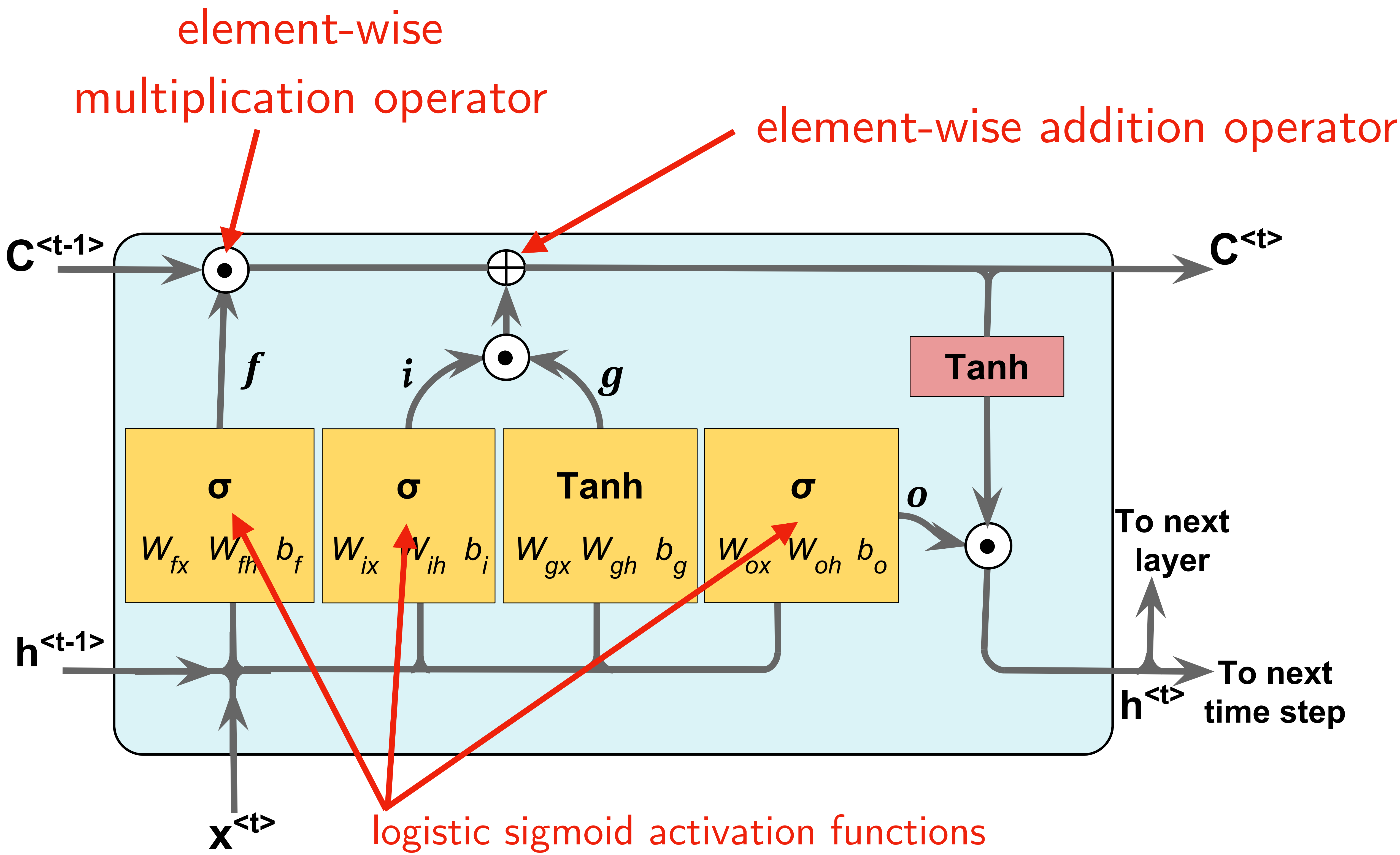
Cell state at current time step



Long-short term memory (LSTM)



Long-short term memory (LSTM)

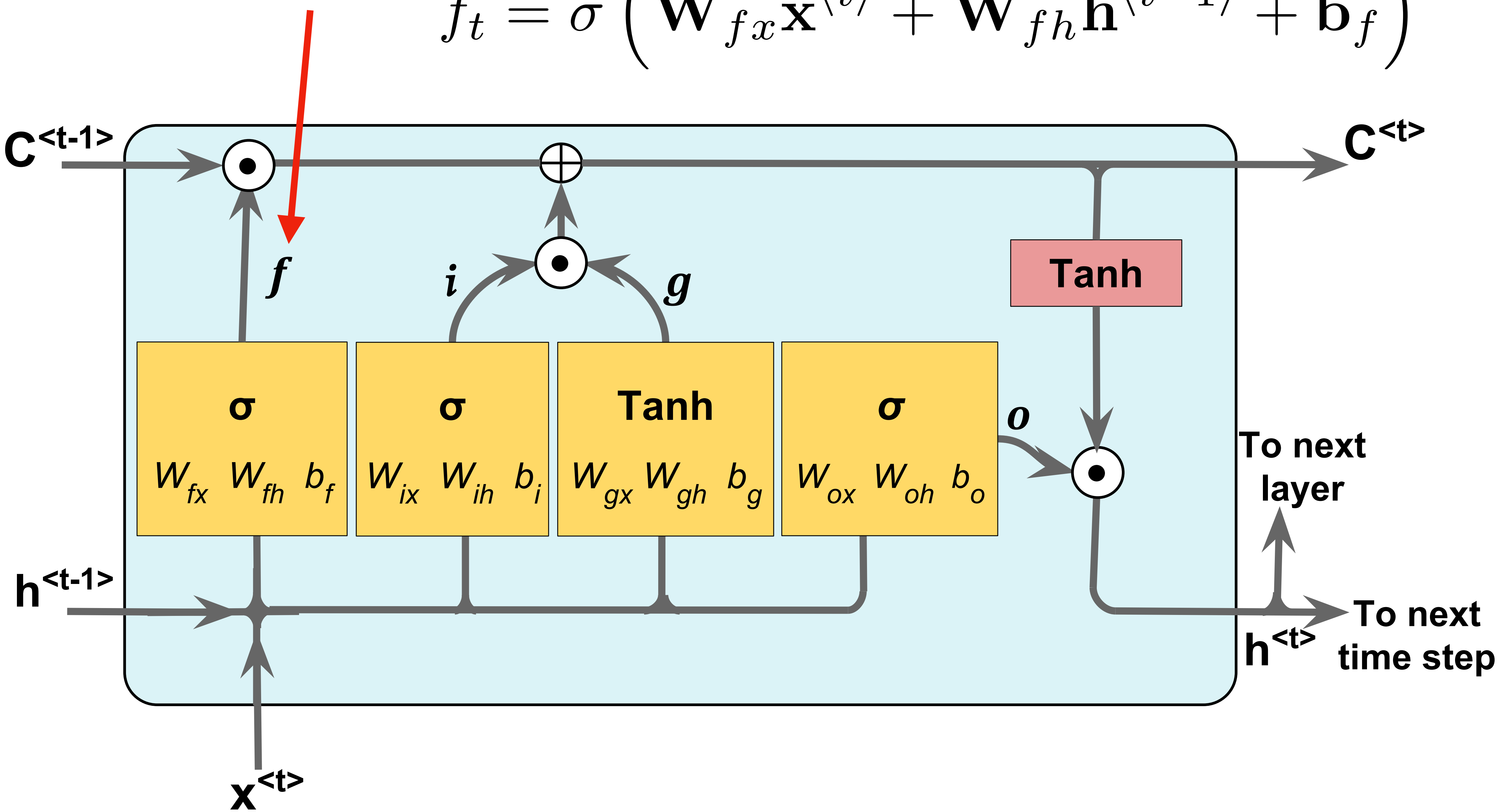


Long-short term memory (LSTM)

Gers, Felix A., Jürgen Schmidhuber, and Fred Cummins. "Learning to forget: Continual prediction with LSTM." (1999): 850-855.

"Forget Gate": controls which information is remembered, and which is forgotten; can reset the cell state

$$f_t = \sigma\left(\mathbf{W}_{fx}\mathbf{x}^{\langle t\rangle} + \mathbf{W}_{fh}\mathbf{h}^{\langle t-1\rangle} + \mathbf{b}_f\right)$$

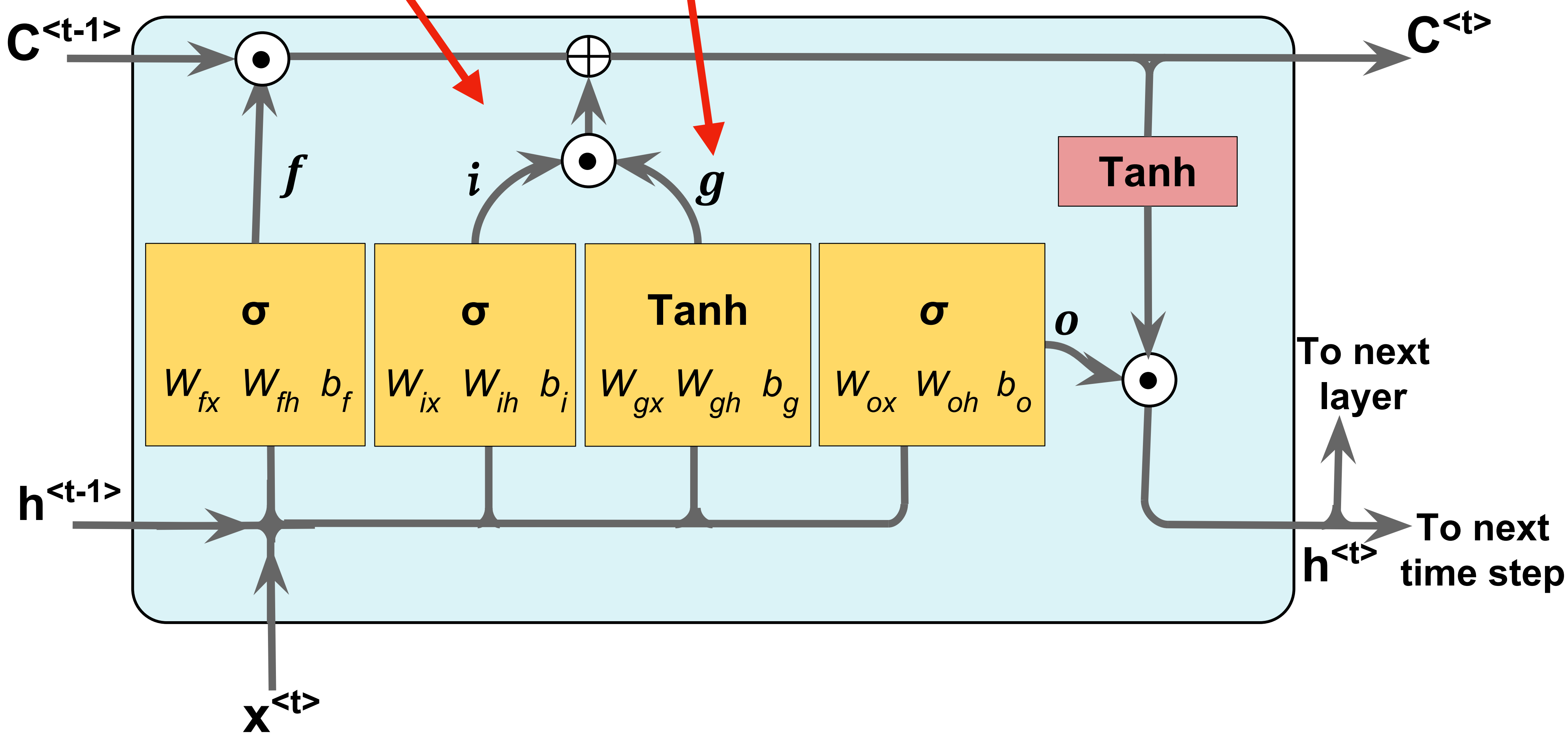


Long-short term memory (LSTM)

"Input Gate": $\mathbf{i}_t = \sigma\left(\mathbf{W}_{ix}\mathbf{x}^{\langle t\rangle} + \mathbf{W}_{ih}\mathbf{h}^{\langle t-1\rangle} + \mathbf{b}_i\right)$

"Input Node":

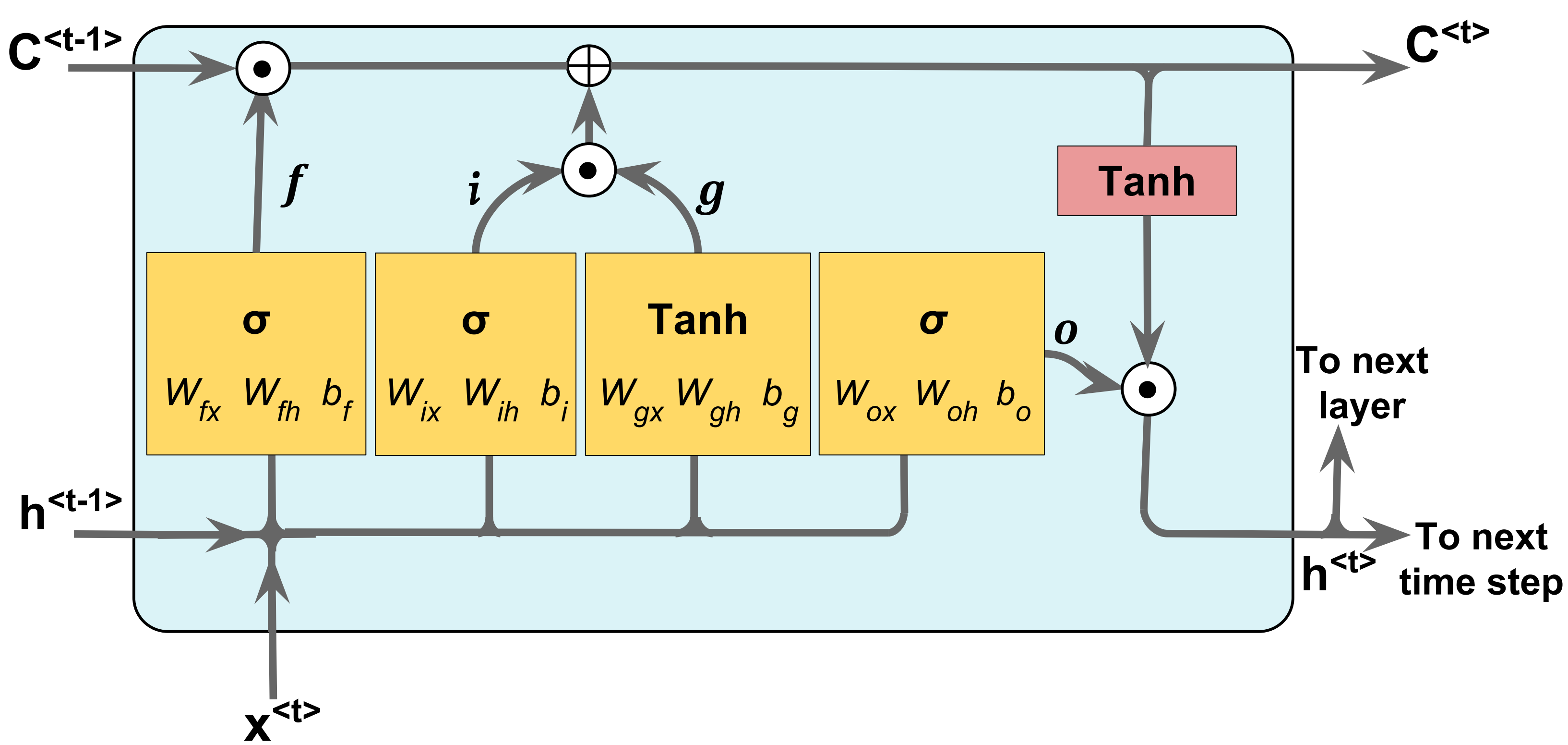
$$\mathbf{g}_t = \tanh\left(\mathbf{W}_{gx}\mathbf{x}^{\langle t\rangle} + \mathbf{W}_{gh}\mathbf{h}^{\langle t-1\rangle} + \mathbf{b}_g\right)$$



Long-short term memory (LSTM)

Brief summary of the gates so far ...

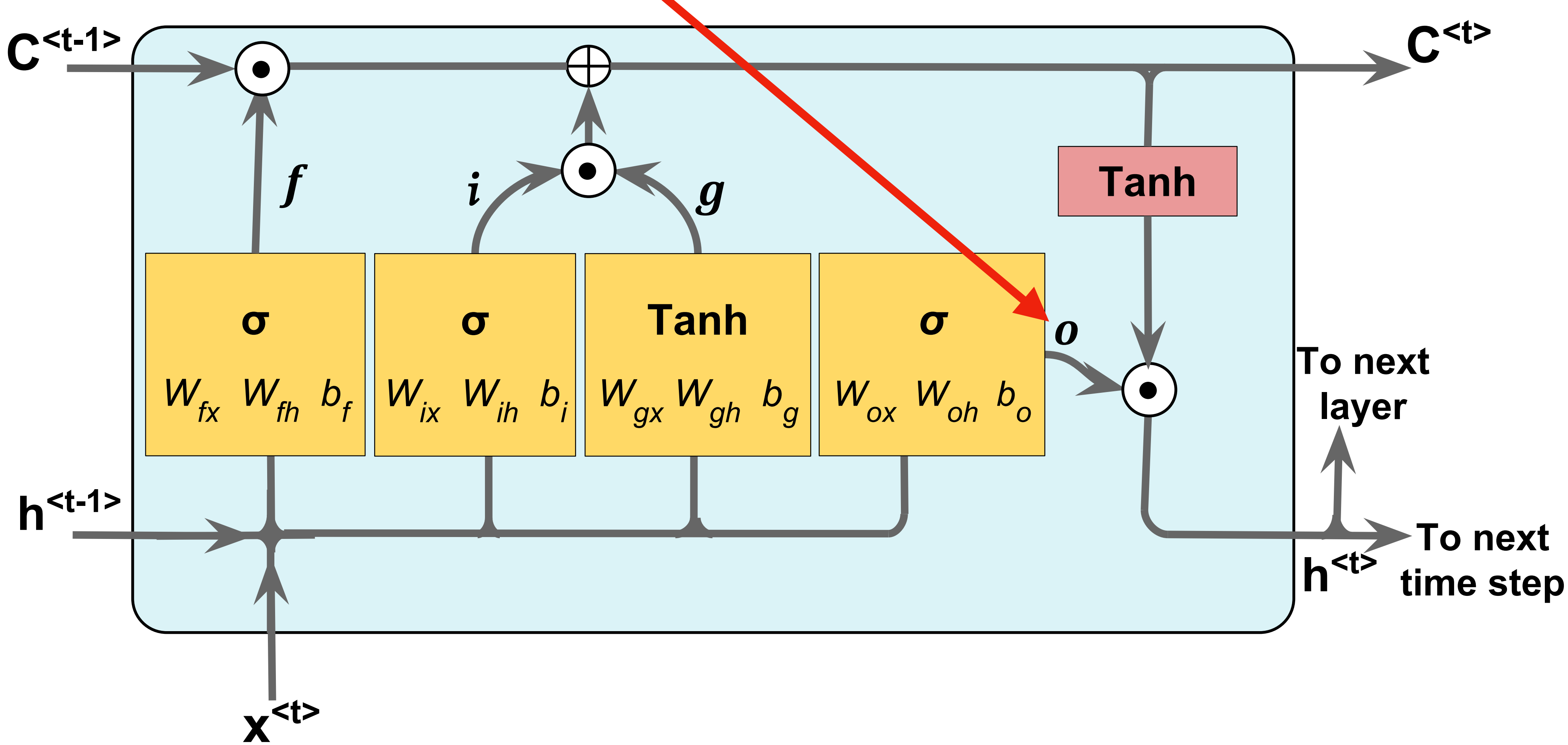
$$C^{\langle t \rangle} = \left(C^{\langle t-1 \rangle} \odot \overset{\text{Forget Gate}}{f_t} \right) \oplus \underbrace{\left(\overset{\text{Input Node}}{i_t} \odot \overset{\text{Input Gate}}{g_t} \right)}_{\text{For updating the cell state}}$$



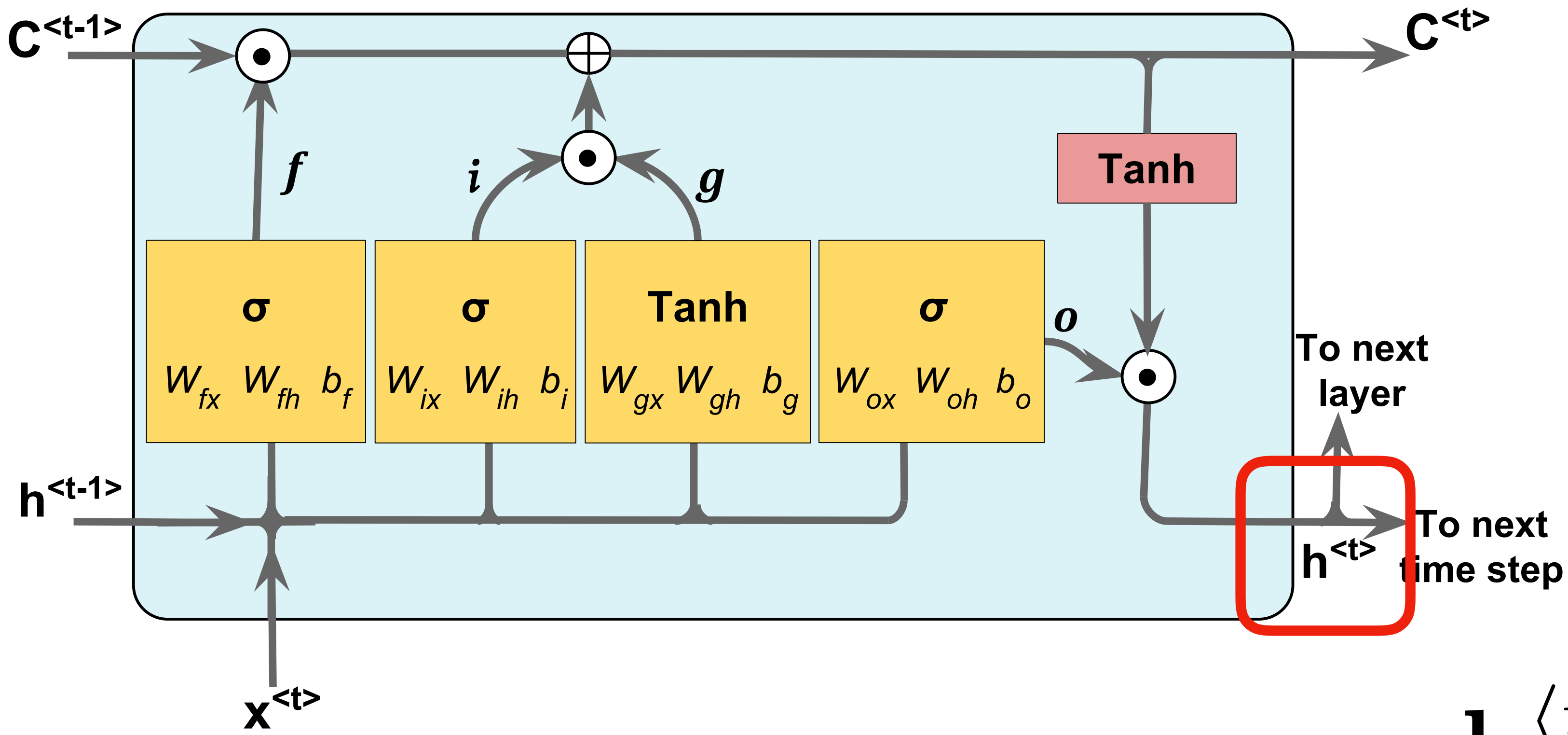
Long-short term memory (LSTM)

Output gate for updating the values of hidden units:

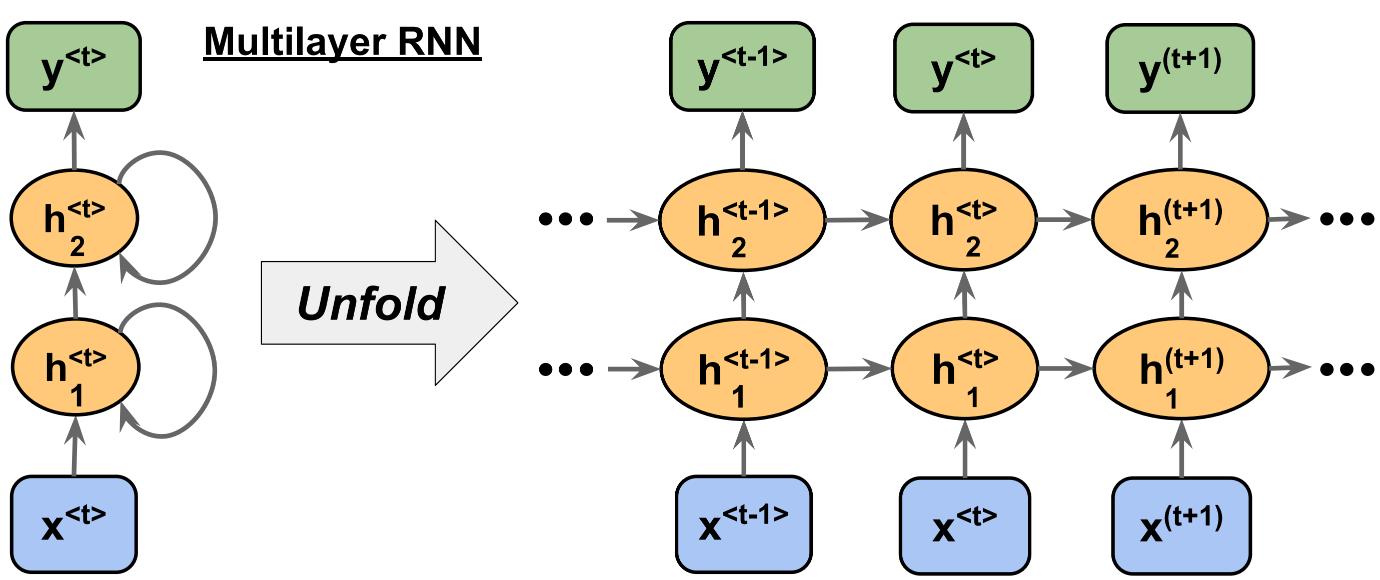
$$\mathbf{o}_t = \sigma\left(\mathbf{W}_{ox}\mathbf{x}^{\langle t\rangle} + \mathbf{W}_{oh}\mathbf{h}^{\langle t-1\rangle} + \mathbf{b}_o\right)$$



Long-short term memory (LSTM)



$$\mathbf{h}^{\langle t \rangle} = \mathbf{o}_t \odot \tanh\left(\mathbf{C}^{\langle t \rangle}\right)$$

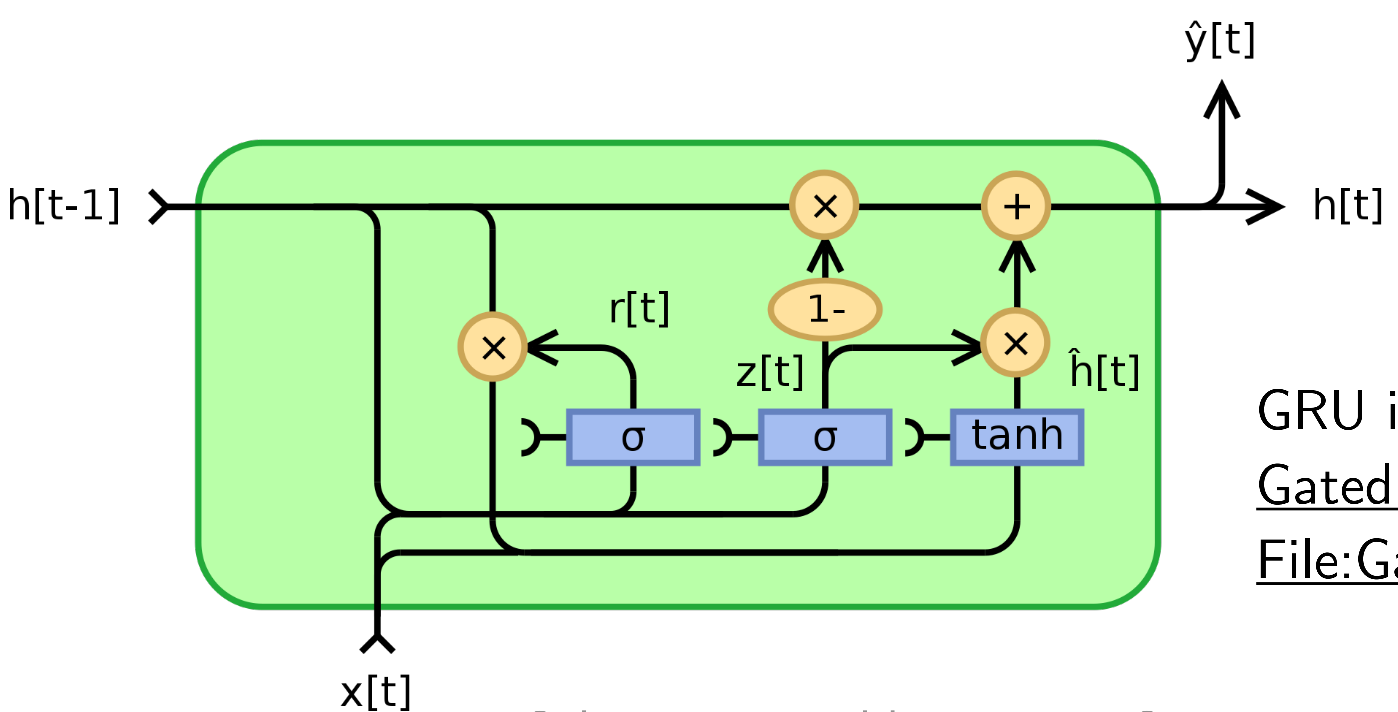


Long-short term memory (LSTM)

- Still popular and widely used today
- A recent, related approach is the Gated Recurrent Unit (GRU)

  Cho, Kyunghyun, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. "Learning phrase representations using RNN encoder-decoder for statistical machine translation." *arXiv preprint arXiv:1406.1078* (2014).
- Nice article exploring LSTMs and comparing them to GRUs

  Jozefowicz, Rafal, Wojciech Zaremba, and Ilya Sutskever. "An empirical exploration of recurrent network architectures." In *International Conference on Machine Learning*, pp. 2342-2350. 2015.



GRU image source: https://en.wikipedia.org/wiki/Gated_recurrent_unit#/media/File:Gated_Recurrent_Unit,_base_type.svg

RNNs with LSTMs in PyTorch

Conceptually simple, the (very) hard part is the data processing pipeline

```
...
    self.rnn = torch.nn.LSTMCell(input_size, hidden_size)
...


def forward(self, x):
    embedded = self.embedding(text)
    h = self.initial_hidden_state()
    for input in x:
        h = self.rnn(input, h)
```

```
...
    self.rnn = torch.nn.LSTM(input_size, hidden_size)
...


def forward(self, x):
    h_0 = self.initial_hidden_state()
    output, h = self.rnn(x, h_0)
```

These two are equivalent, but the bottom one is substantially faster

Different Types of Sequence Modeling Tasks

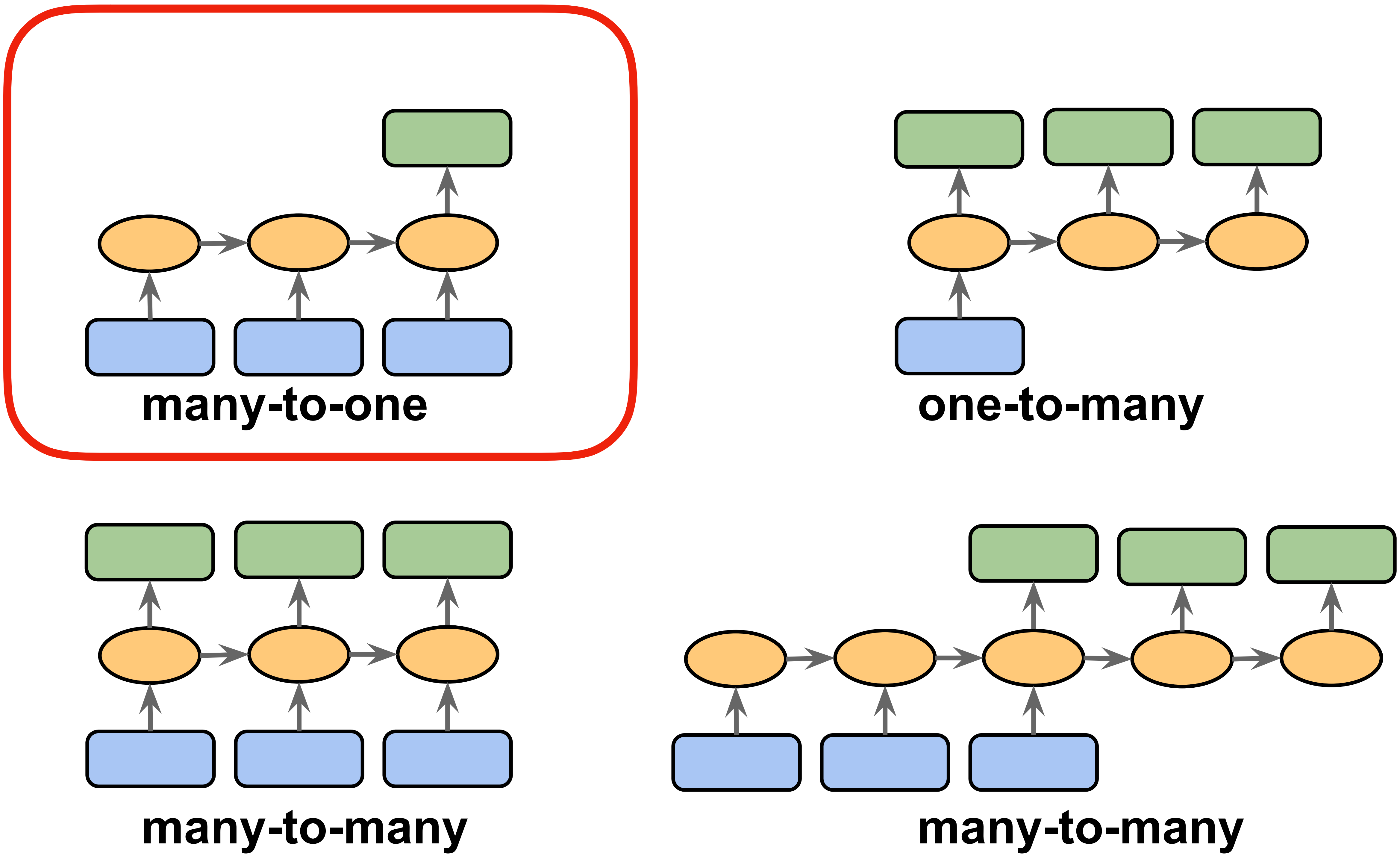
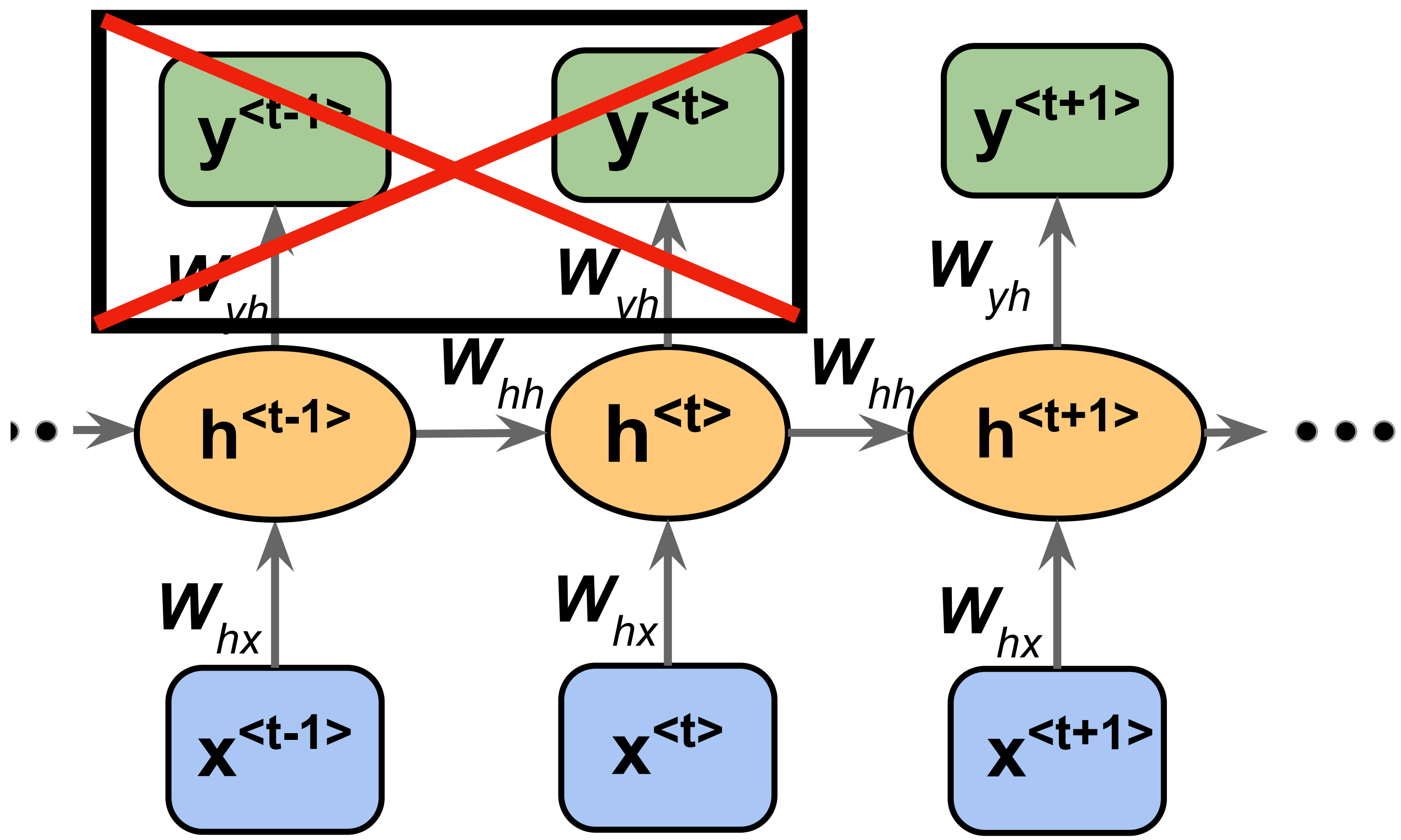


Figure based on:

The Unreasonable Effectiveness of Recurrent Neural Networks by Andrej Karpathy (http://karpathy.github.io/2015/05/21/rnn-effectiveness/)

Many-to-One



bentrevett / pytorch-sentiment-analysis

Watch 25 | Unstar 573 | Fork 116

Code | Issues 0 | Pull requests 0 | Projects 1 | Wiki | Insights

Tutorials on getting started with PyTorch and TorchText for sentiment analysis.

pytorch | sentiment-analysis | tutorial | rnn | lstm | fasttext | torchtext | sentiment-classification | cnn | cnn-text-classification | lstm-sentiment-analysis | pytorch-tutorial

65 commits | 1 branch | 0 releases | 2 contributors | MIT

Branch: master | New pull request | Create new file | Upload files | Find File | Clone or download

bentrevett removed unnecessary torch.cuda.manual_seed from appendix c notebook — Latest commit `61093d8` 19 hours ago

assets	fixed wording and updated assets for notebook 4	a month ago
custom_embeddings	added appendix c - handling embeddings	15 days ago
data	added optional appendix for how to use your own dataset with torchtext	11 months ago
.gitignore	removed testing cell from notebook C, updated gitignore to ignore tra...	19 hours ago
1 - Simple Sentiment Analysis.ipynb	added model.eval() in predict sentiment functions (#31)	7 days ago
2 - Upgraded Sentiment Analysis.i...	added model.eval() in predict sentiment functions (#31)	7 days ago
3 - Faster Sentiment Analysis.ipynb	added model.eval() in predict sentiment functions (#31)	7 days ago
4 - Convolutional Sentiment Analy...	added model.eval() in predict sentiment functions (#31)	7 days ago
5 - Multi-class Sentiment Analysis....	added model.eval() in predict sentiment functions (#31)	7 days ago
A - Using TorchText with Your Own...	mention how using sort key is preferred over sort = False	25 days ago
B - A Closer Look at Word Embedd...	updated appendix B - formatting and typos	16 days ago

Many-to-One ("Word"-RNN)

1 - Simple Sentiment Analysis (Simple RNN)
https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/1%20-%20Simple%20Sentiment%20Analysis.ipynb

2 - Updated Sentiment Analysis (LSTM)
https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/2%20-%20Upgraded%20Sentiment%20Analysis.ipynb

Optional:

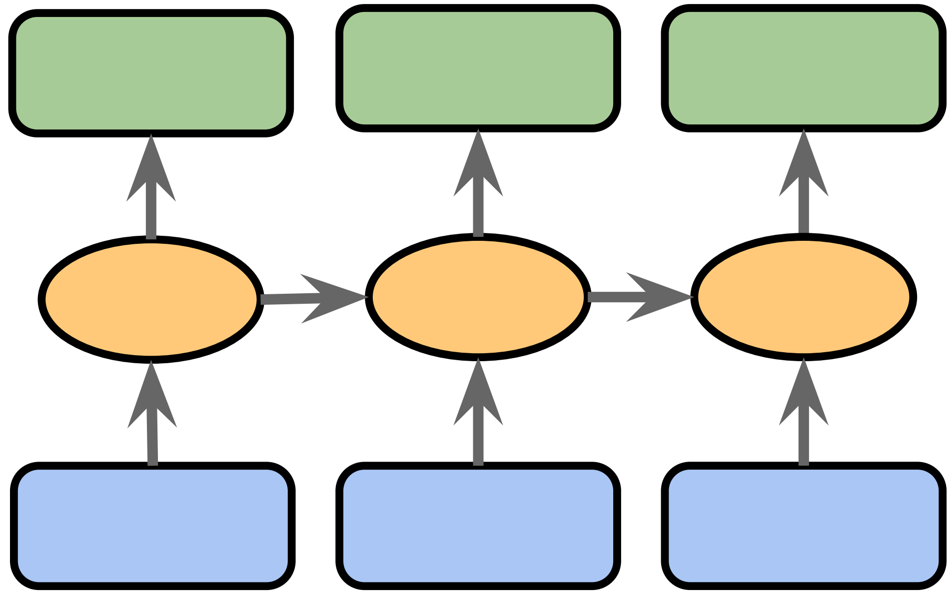
Using TorchText with Your Own Datasets
https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/A%20-%20Using%20TorchText%20with%20Your%20Own%20Datasets.ipynb

4 - Convolutional Sentiment Analysis
https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/4%20-%20Convolutional%20Sentiment%20Analysis.ipynb

Many-to-One ("Character"-RNN)

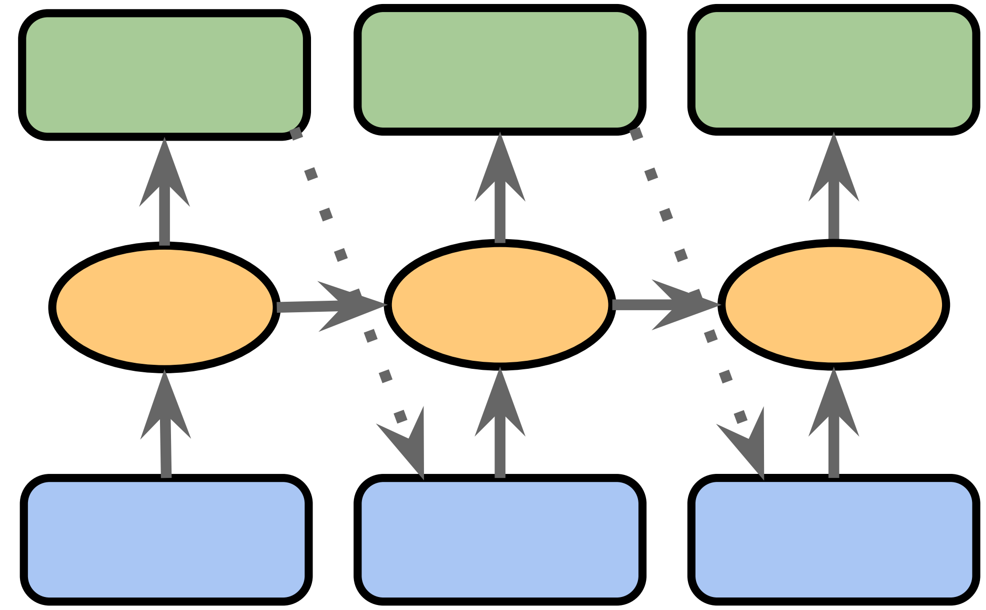
Classifying Names with a Character-level RNN

https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html



many-to-many

"training"



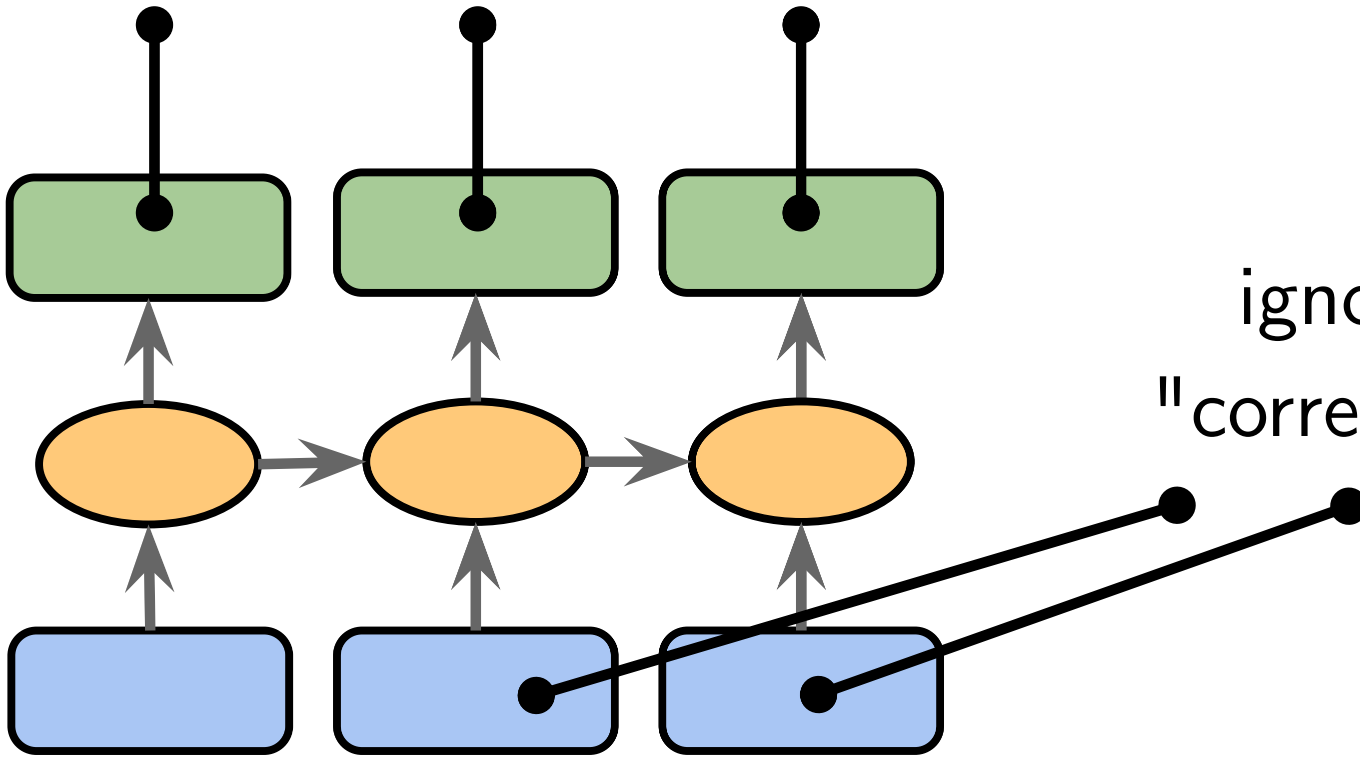
~~many~~-to-many

"one"

"generating new text"

Generating Names with a Character-level RNN
https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

At each time step
Softmax output (probability)
for each possible"next letter"

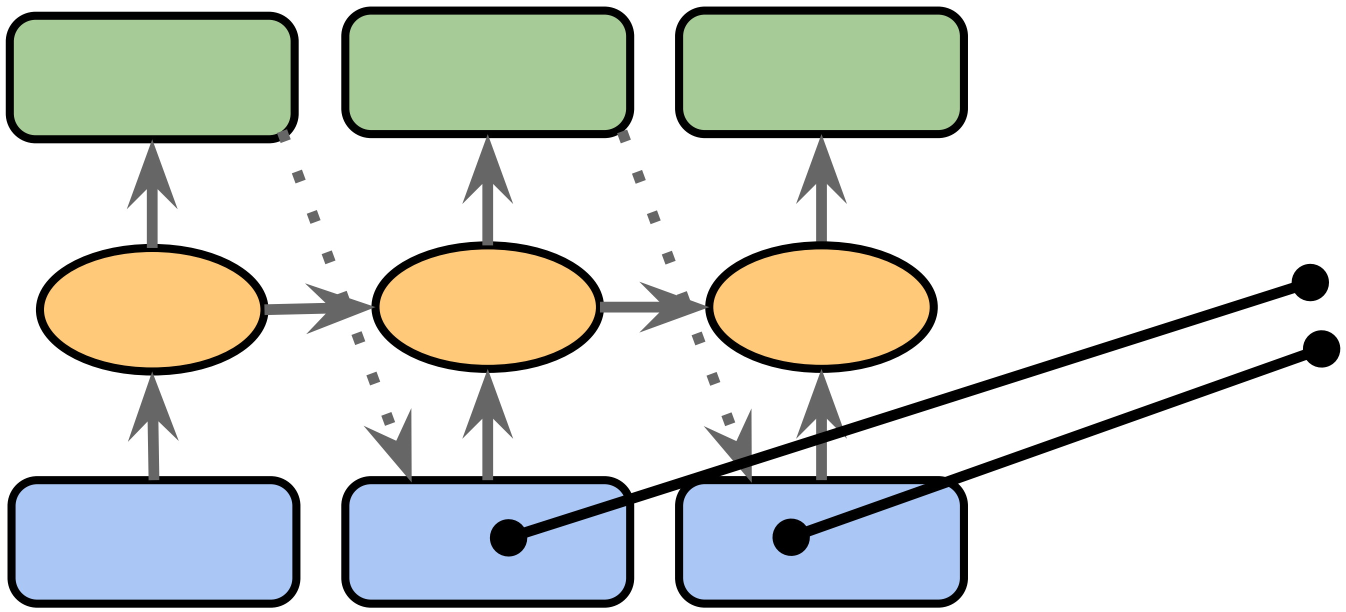


For next each input,
ignore the prediction but use the
"correct" next letter from the dataset

many-to-many

"training"

Generating Names with a Character-level RNN
https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html



To generate new text, now, sample from the softmax outputs and provide the letter as input to the next time step

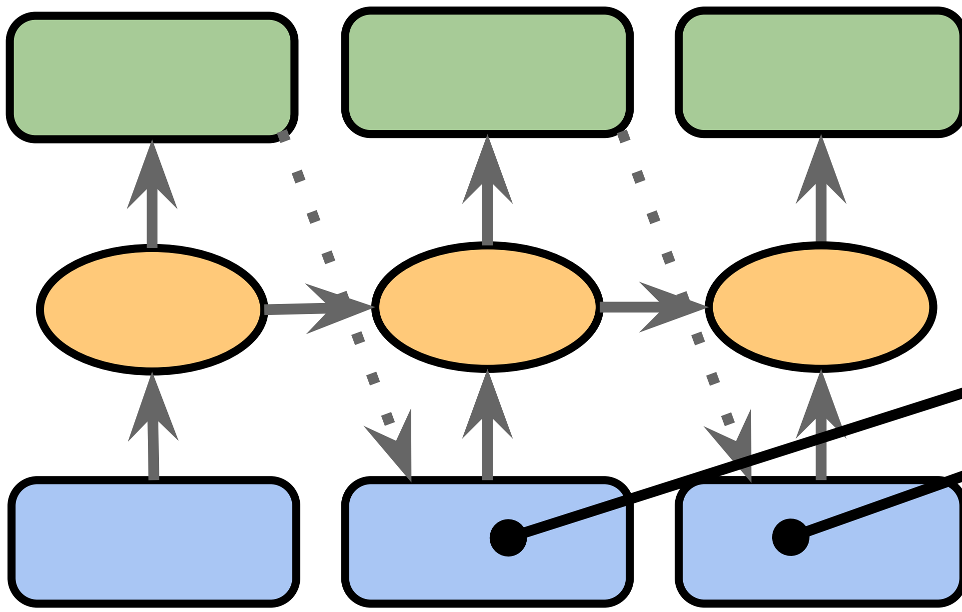
~~many~~-to-many

"one"

"generating new text"

Generating Names with a Character-level RNN

https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html



~~many~~-to-many

"one"

"generating new text"

To generate new text, now, sample from the softmax outputs and provide the letter as input to the next time step

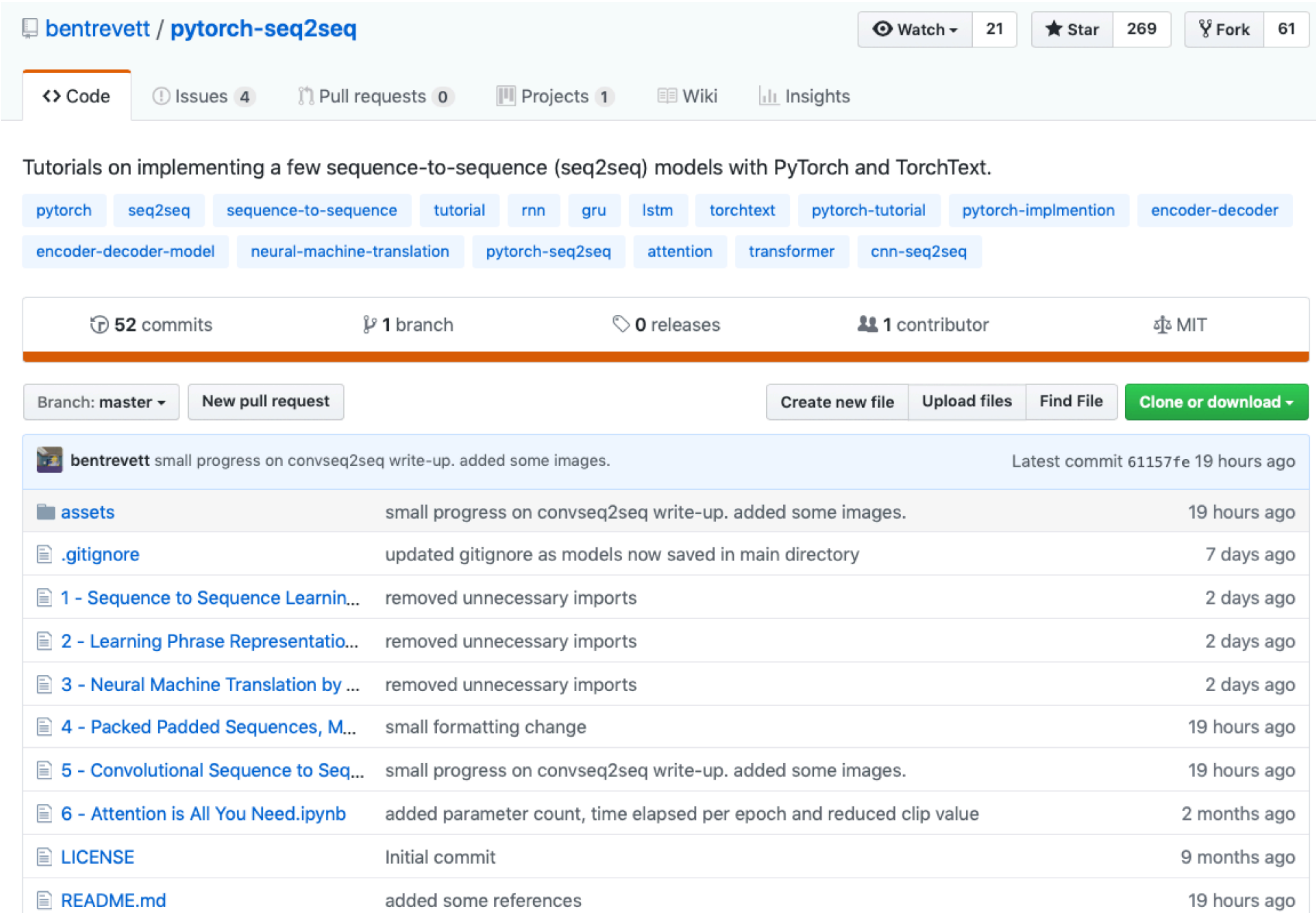
Note that this approach works with both Word- and Character-RNNs

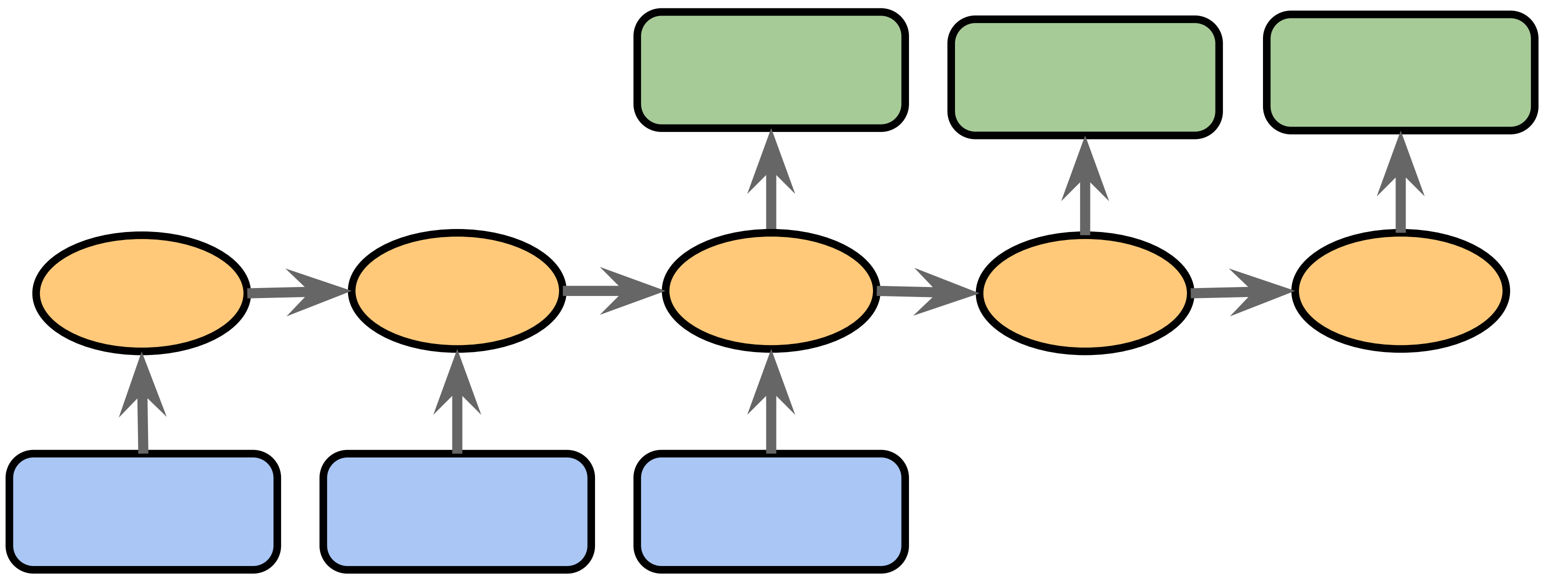
Generating Names with a Character-level RNN

https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

Additional Tutorials:

https://github.com/bentrevett/pytorch-seq2seq





many-to-many

Translation with a Sequence to Sequence Network and Attention (English to French)

https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html

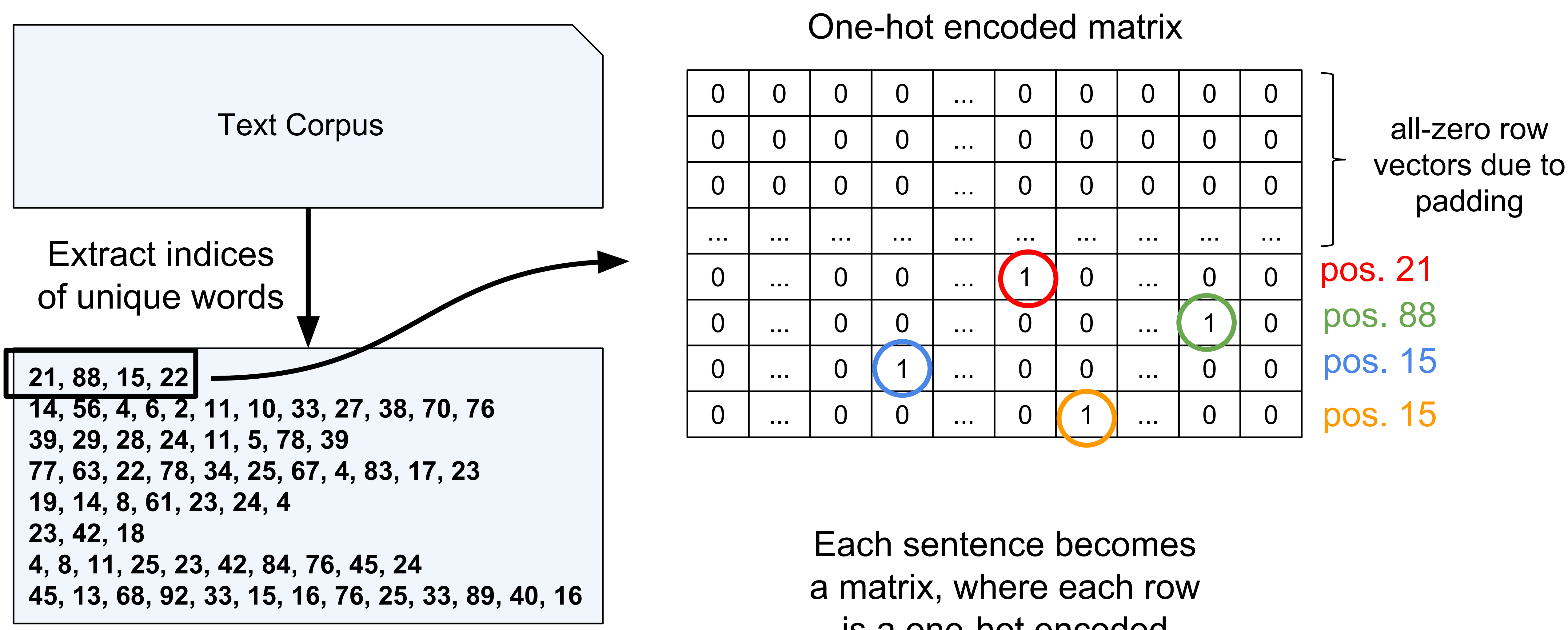
Figure based on:

The Unreasonable Effectiveness of Recurrent Neural Networks by Andrej Karpathy (http://karpathy.github.io/2015/05/21/rnn-effectiveness/)

Data Processing for a Word RNN (e.g., for sentiment classification)

Data Processing for Word-RNNs

Step 1): Create one-hot encoded matrix



Each sentence becomes a matrix, where each row is a one-hot encoded vector

Data Processing for Word-RNNs

Step 2): Multiply with embedding matrix

One-hot encoded matrix

0	0	0	0	...	0	0	0	0	0
0	0	0	0	...	0	0	0	0	0
0	0	0	0	...	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...
0	...	0	0	...	1	0	...	0	0
0	...	0	0	...	0	0	...	1	0
0	...	0	1	...	0	0	...	0	0
0	...	0	0	...	0	1	...	0	0

all-zero row vectors due to padding

pos. 21
pos. 88
pos. 15
pos. 15

Each sentence becomes a matrix, where each row is a one-hot encoded vector

A trainable matrix of type real

$r_{1,1}$	$r_{1,2}$	$r_{1,3}$	$r_{1,4}$	$r_{1,5}$	$r_{1,6}$
$r_{2,1}$	$r_{2,2}$	$r_{2,3}$	$r_{2,4}$	$r_{2,5}$	$r_{2,6}$
$r_{3,1}$	$r_{3,2}$	$r_{3,3}$	$r_{3,4}$	$r_{3,5}$	$r_{3,6}$
$r_{4,1}$	$r_{4,2}$	$r_{4,3}$	$r_{4,4}$	$r_{4,5}$	$r_{4,6}$
$r_{5,1}$	$r_{5,2}$	$r_{5,3}$	$r_{5,4}$	$r_{5,5}$	$r_{5,6}$
$r_{6,1}$	$r_{6,2}$	$r_{6,3}$	$r_{6,4}$	$r_{6,5}$	$r_{6,6}$
$r_{7,1}$	$r_{7,2}$	$r_{7,3}$	$r_{7,4}$	$r_{7,5}$	$r_{7,6}$
$\vdots$					
$r_{n-2,1}$	$r_{n-2,2}$	$r_{n-2,3}$	$r_{n-2,4}$	$r_{n-2,5}$	$r_{n-2,6}$
$r_{n-1,1}$	$r_{n-1,2}$	$r_{n-1,3}$	$r_{n-1,4}$	$r_{n-1,5}$	$r_{n-1,6}$
$r_{n,1}$	$r_{n,2}$	$r_{n,3}$	$r_{n,4}$	$r_{n,5}$	$r_{n,6}$

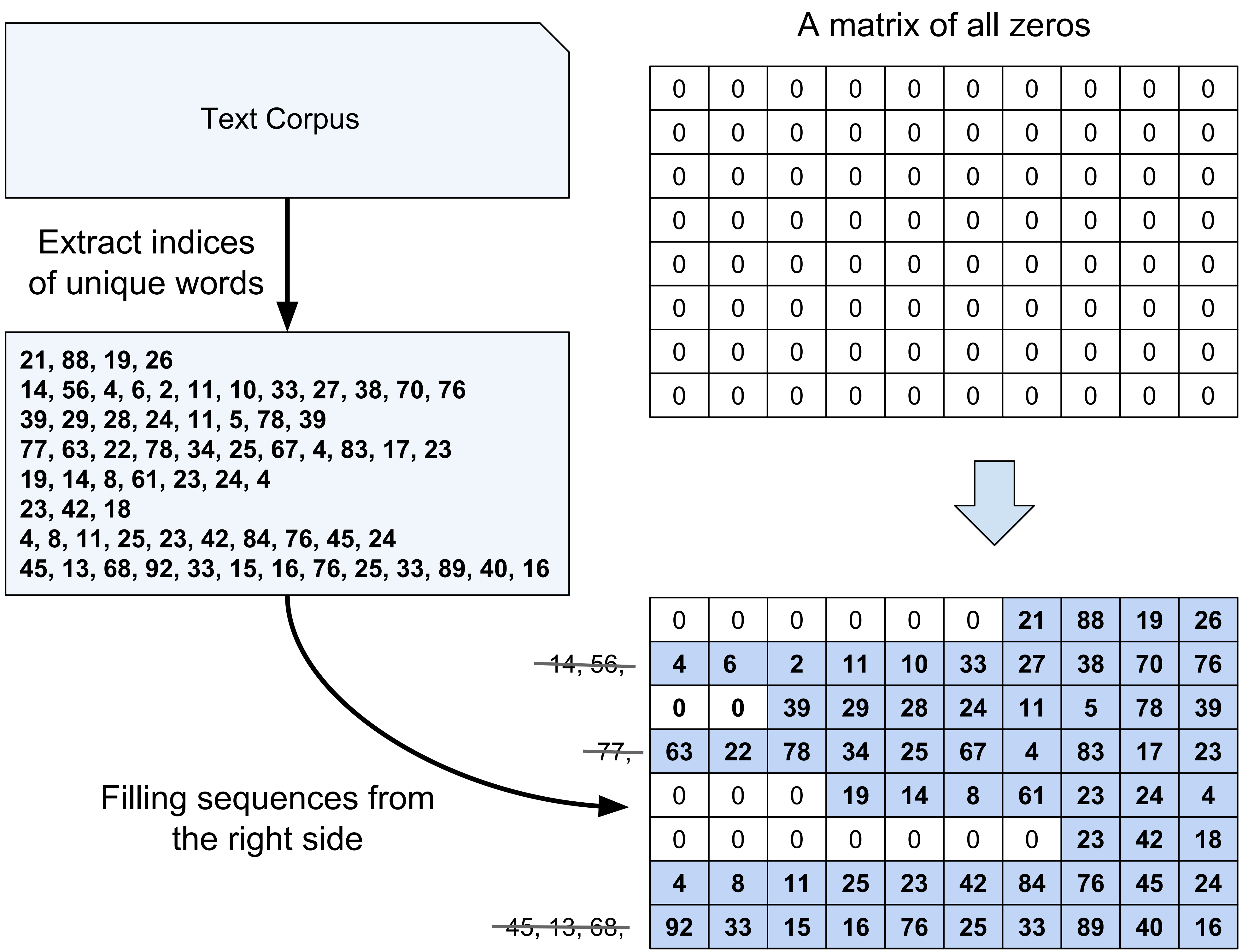
Number of unique words (or n_{words})

Number of features (or *embedding size*)

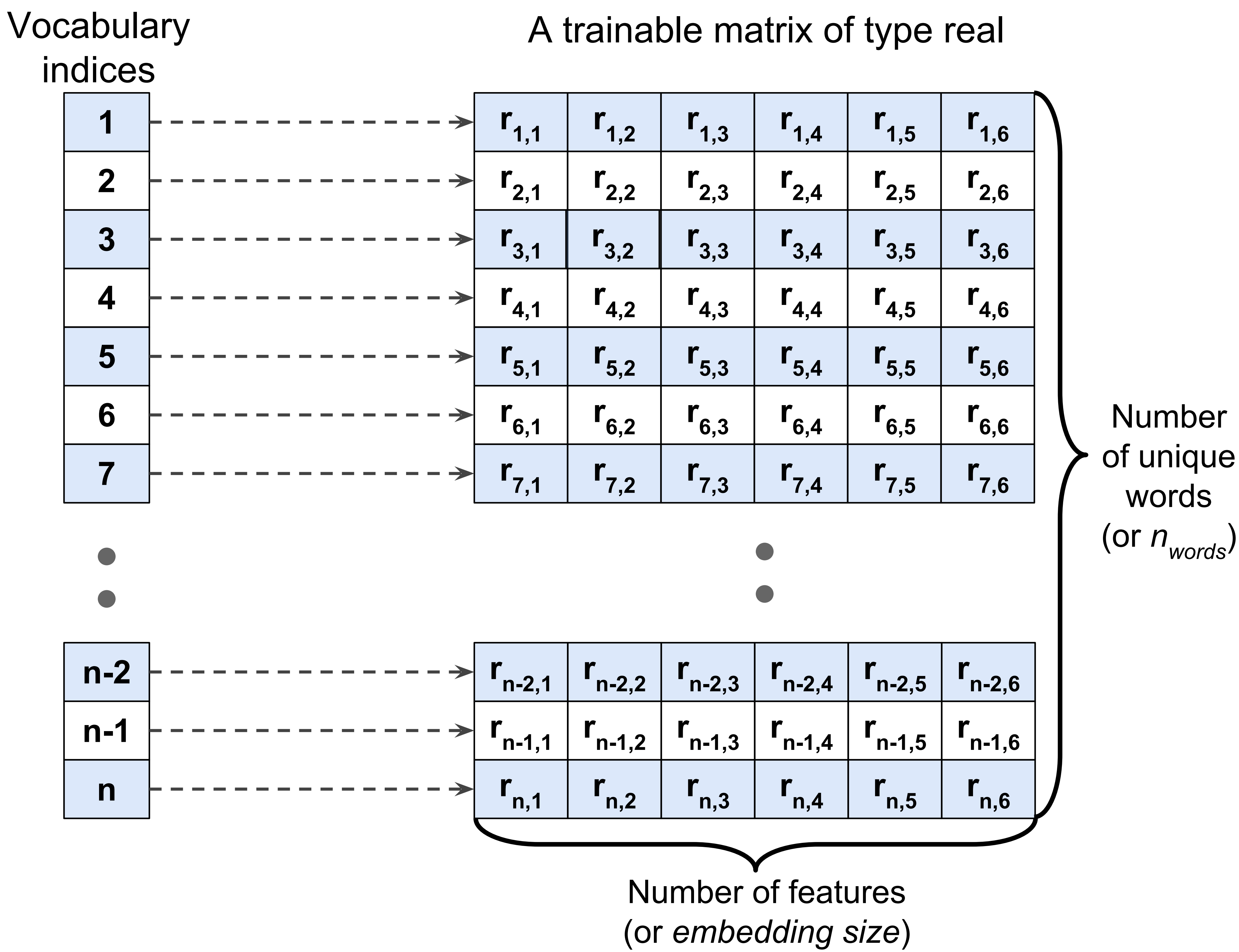
Output is a matrix $\in \mathbb{R}^{\text{num\_words} \times \text{vocab\_size}}$

The "huge" matrix multiplication is very inefficient, so we replace it with an embedding "look-up"

Step 1: Read Sentences into a Word Matrix

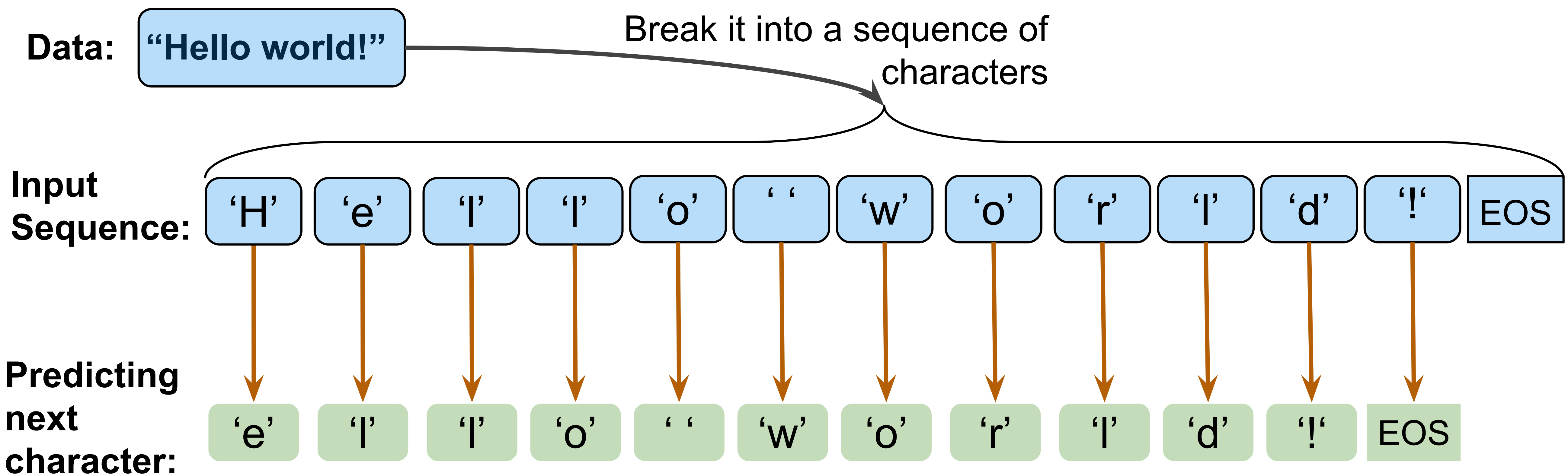


Step 2: "Look up" row that correspond to word index

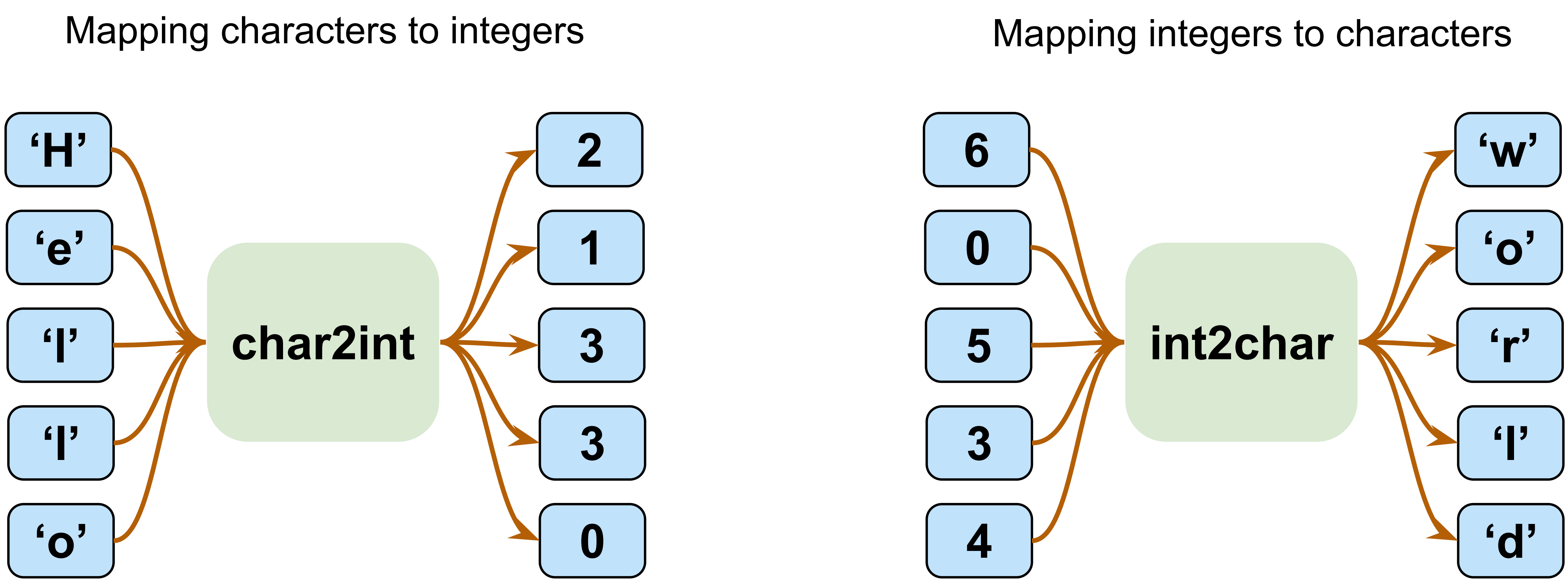


Data Processing for a Character RNN (e.g., for text generation)

Step 1: Break up text into characters

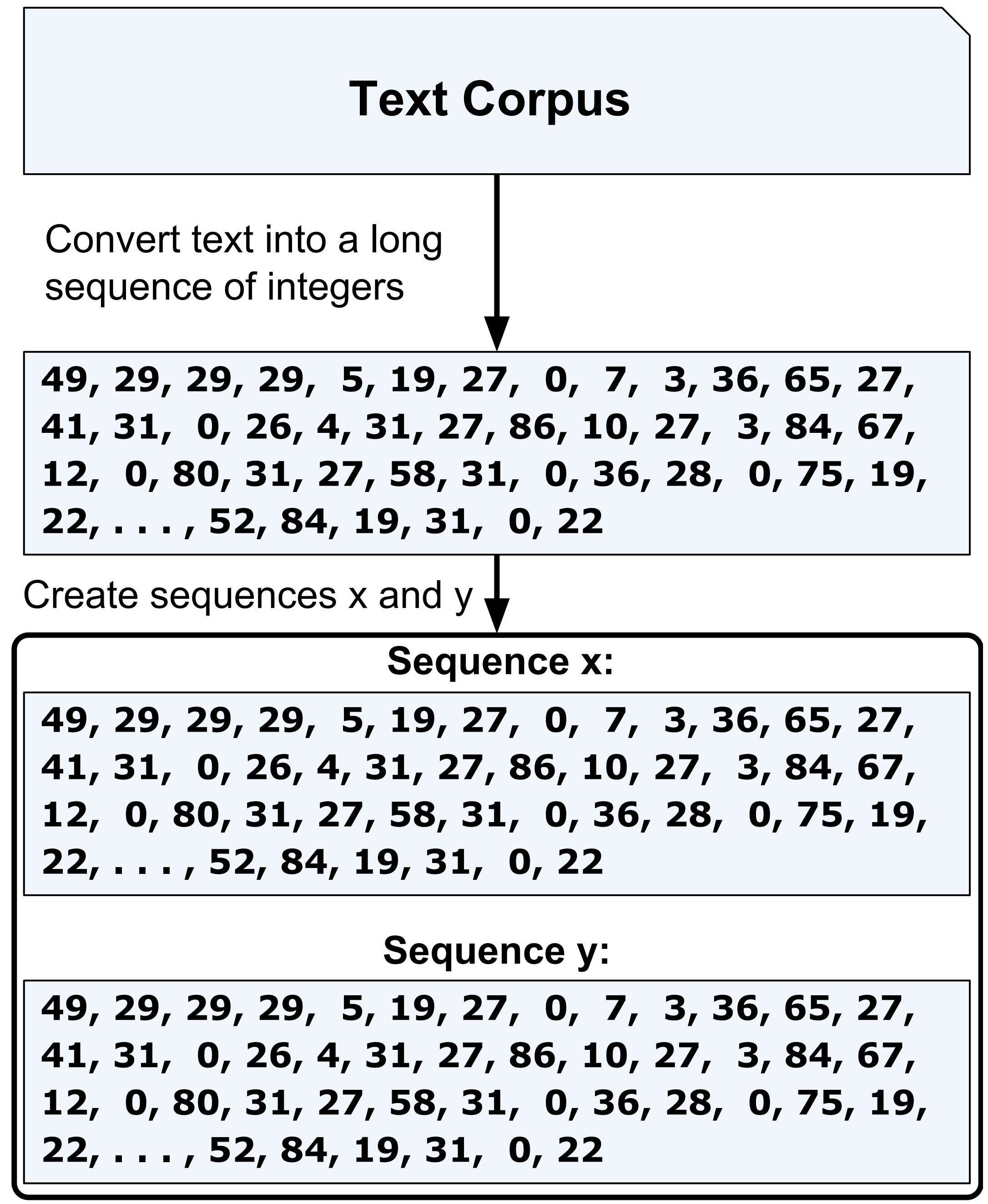


Step 2: Define Mapping Dictionaries



Step 3: Define Inputs and Outputs

Outputs are the characters shifted by 1 position as we want to predict the next character



More details for how this is specifically implement in PyTorch are shown in the excellent Jupyter Notebooks I linked in the earlier slides