TTIC 31190: Natural Language Processing

Kevin Gimpel
Spring 2018

Lecture 10: Recurrent, Recursive, and Convolutional Neural Networks in NLP

Assignment 2 due Monday

- questions?

Project Proposal

- project proposal details have been posted (see main course page or assignments page)
- due May 9
- groups of 2-3 are ok (but think about how you will divide up the work, especially with 3)
- let me know if you're still looking for a partner

Project

- final report due Wednesday, June 6
- for graduating students, due May 30

Roadmap

- words, morphology, lexical semantics
- text classification
- language modeling
- word embeddings
- recurrent/recursive/convolutional networks in NLP
- sequence labeling, HMMs, dynamic programming
- syntax and syntactic parsing
- semantics, compositionality, semantic parsing
- machine translation and other NLP tasks

word2vec Score Functions

- skip-gram:

$$\text{score}(x, y, \boldsymbol{w}) = \mathbf{w}^{(\text{in},x)} \cdot \mathbf{w}^{(\text{out},y)}$$

inputs (*x*)	outputs (*y*)
agriculture	<s>
agriculture	is
agriculture	the

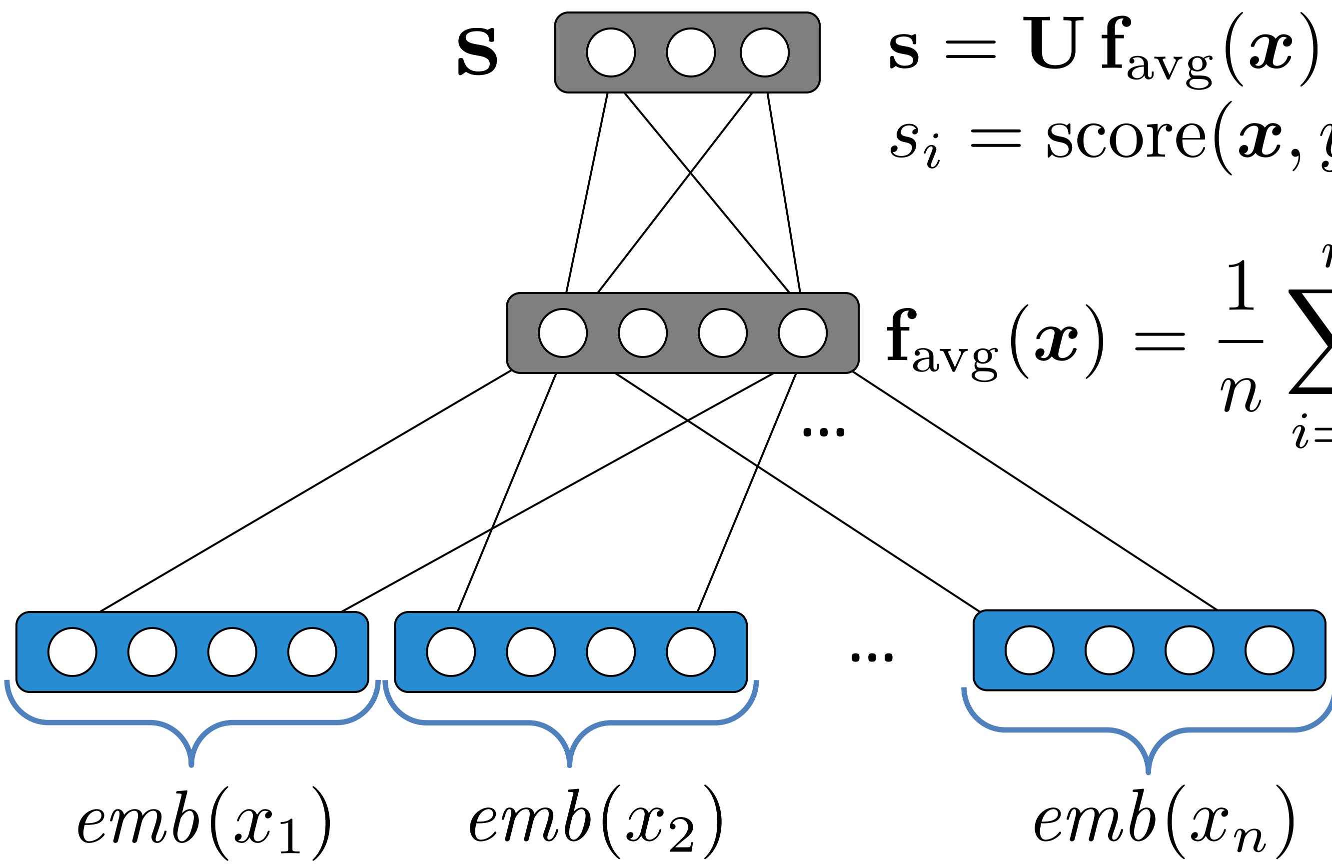
- CBOW:

$$\text{score}(\boldsymbol{x}, y, \boldsymbol{w}) = \left(\frac{1}{|\boldsymbol{x}|} \sum_i \mathbf{w}^{(\text{in},x_i)} \right) \cdot \mathbf{w}^{(\text{out},y)}$$

inputs (*x*)	outputs (*y*)
{<s>, is, the, traditional}	agriculture
{<s>, agriculture, the, traditional}	is
{agriculture, is, traditional, mainstay}	the

A Simple Neural Text Classification Model

- represent ***x*** by averaging its word embeddings
- output is a score vector over all possible labels:



$$\mathbf{s} = \mathbf{U}\, \mathbf{f}_{\mathrm{avg}}(\boldsymbol{x})$$

$$s_i = \mathrm{score}(\boldsymbol{x}, y_i, \boldsymbol{w})$$

$$\mathbf{f}_{\mathrm{avg}}(\boldsymbol{x}) = \frac{1}{n}\sum_{i=1}^{n} emb(x_i)$$

Encoders

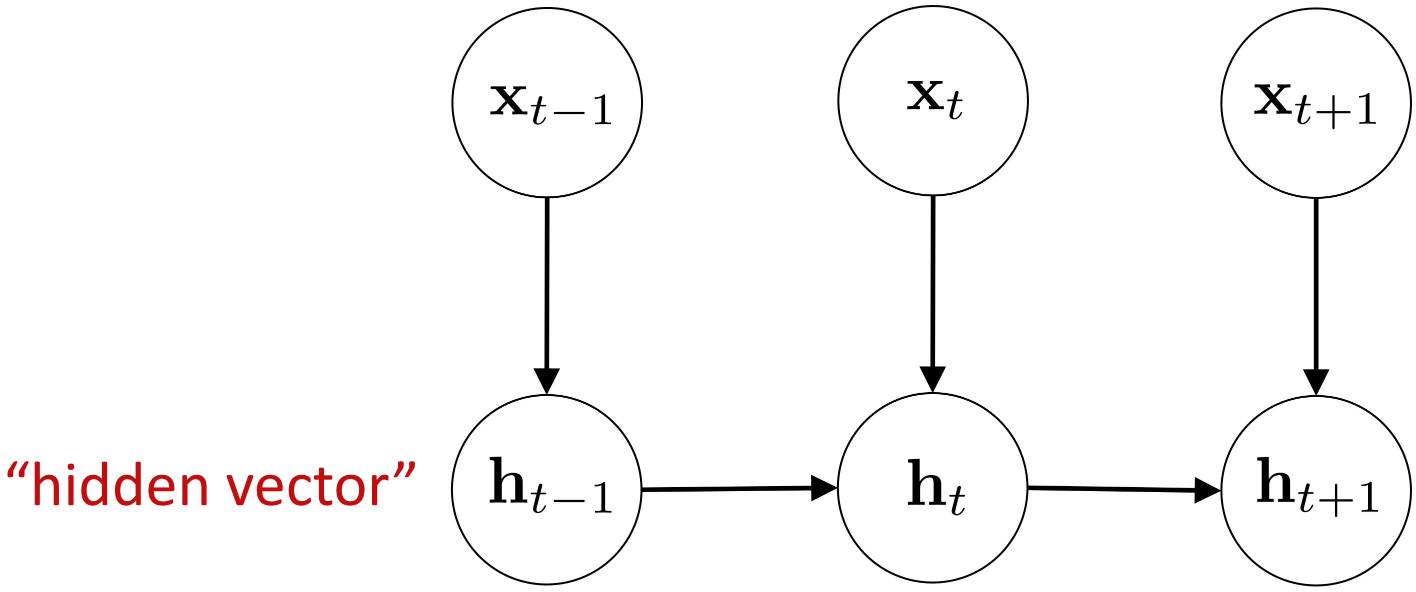
- encoder: a function to represent a word sequence as a vector
- simplest: average word embeddings:

$$\mathbf{f}_{\mathrm{avg}}(\boldsymbol{x}) = \frac{1}{n}\sum_{i=1}^{n} emb(x_i)$$

- many other functions possible!
- lots of recent work on developing better ways to encode word sequences

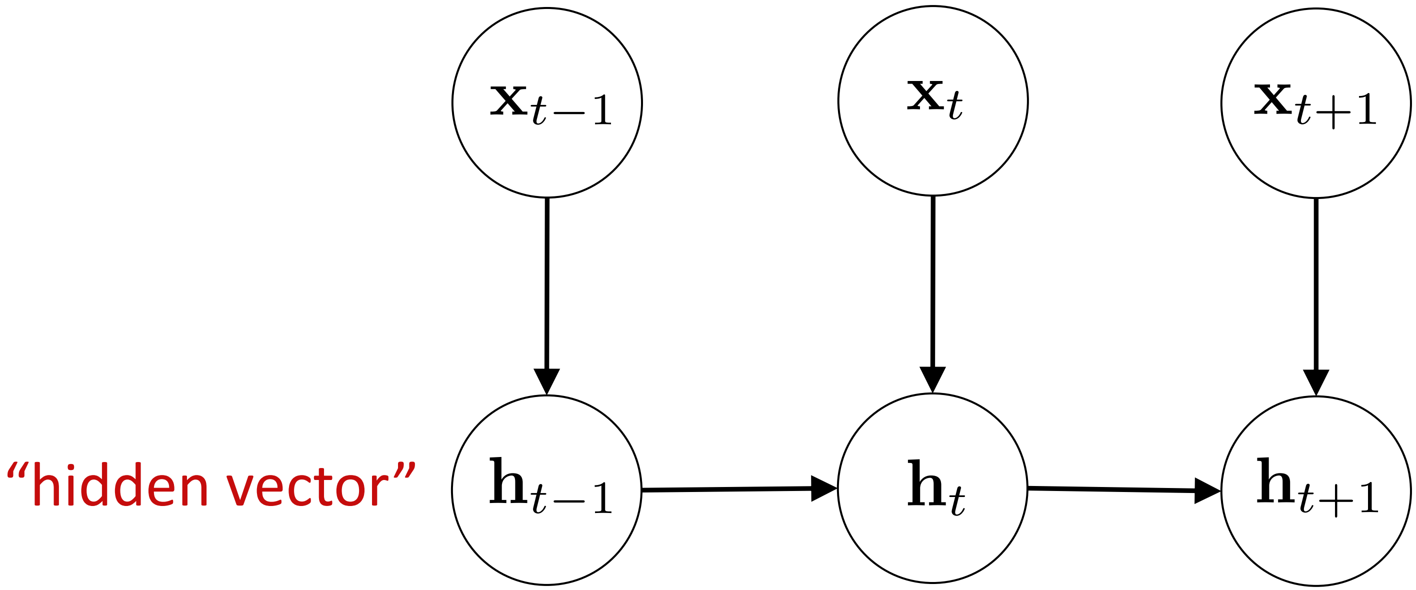
Recurrent Neural Networks

Input is a sequence:



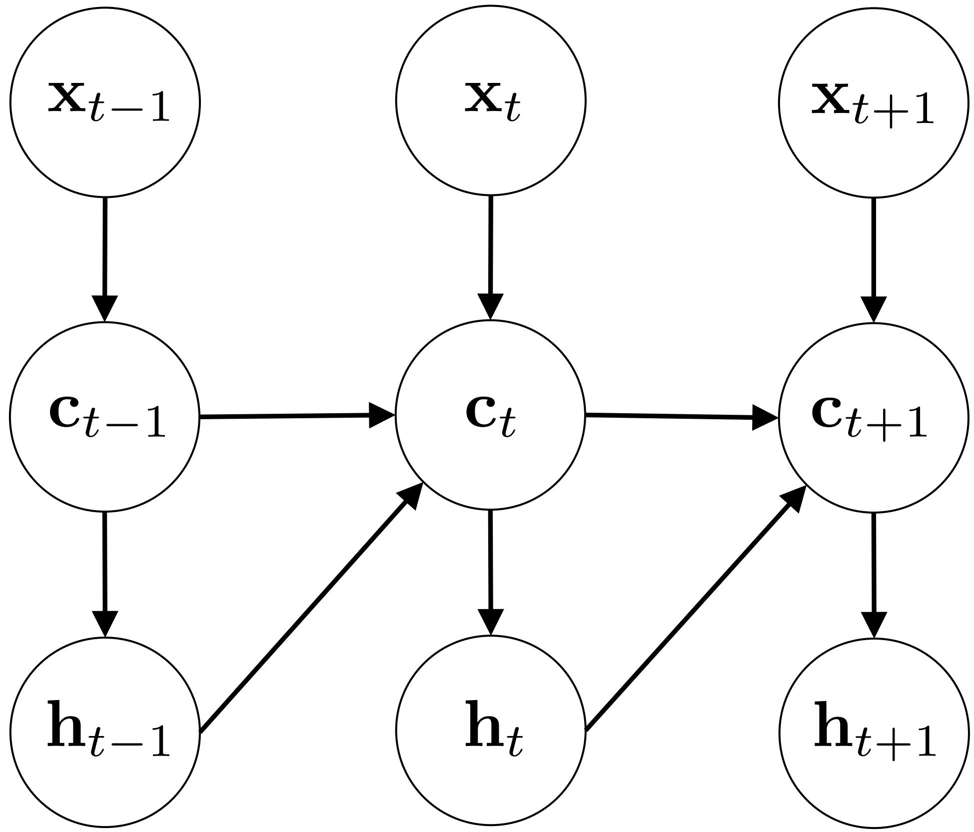
Recurrent Neural Networks

$$\mathbf{h}_t = \tanh\left(\mathbf{W}^{(x)}\mathbf{x}_t + \mathbf{W}^{(h)}\mathbf{h}_{t-1} + \mathbf{b}\right)$$

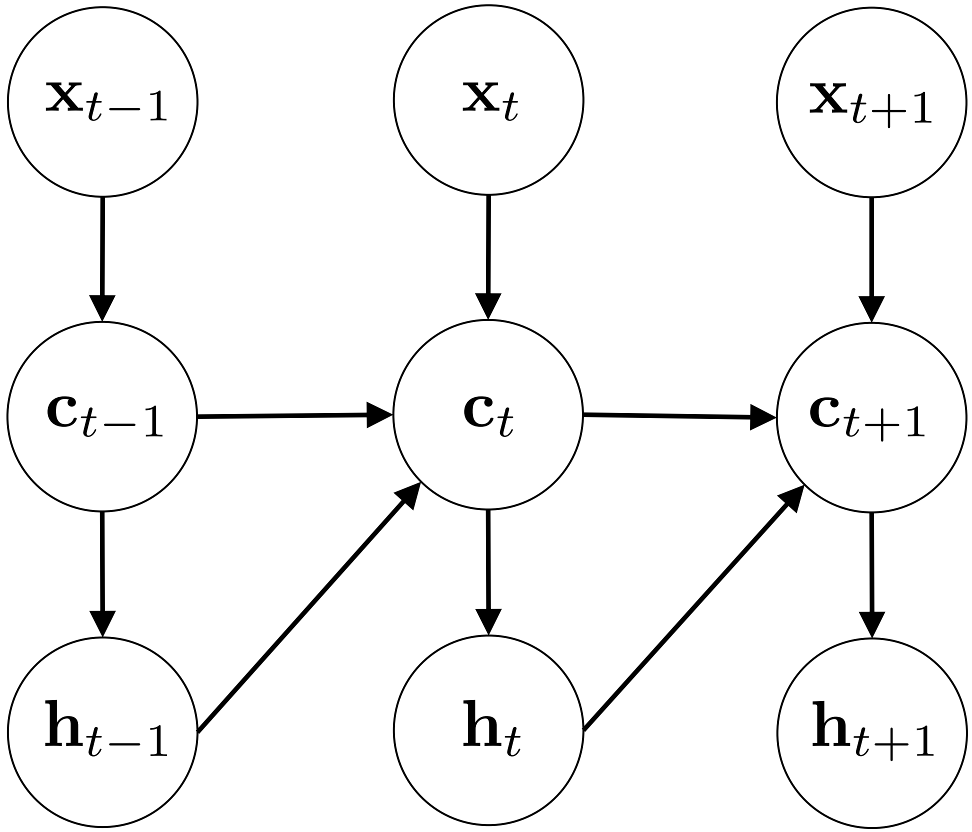


Long Short-Term Memory Networks (gateless)

"memory cell"



Long Short-Term Memory Networks (gateless)

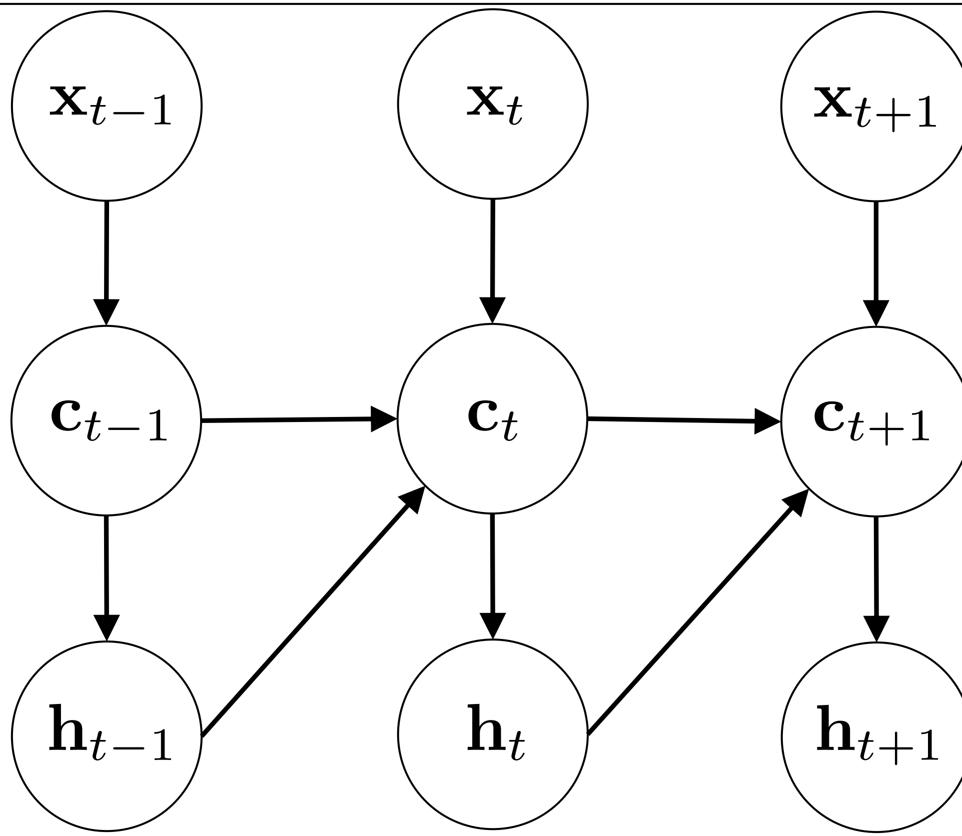


$$\mathbf{h}_t = \tanh(\mathbf{c}_t)$$

Long Short-Term Memory Networks (gateless)

$$\mathbf{c}_t = \mathbf{c}_{t-1} + \tanh\left(\mathbf{W}^{(xc)}\mathbf{x}_t + \mathbf{W}^{(hc)}\mathbf{h}_{t-1} + \mathbf{b}^{(c)}\right)$$

$$\mathbf{h}_t = \tanh(\mathbf{c}_t)$$



Long Short-Term Memory Networks (gateless)

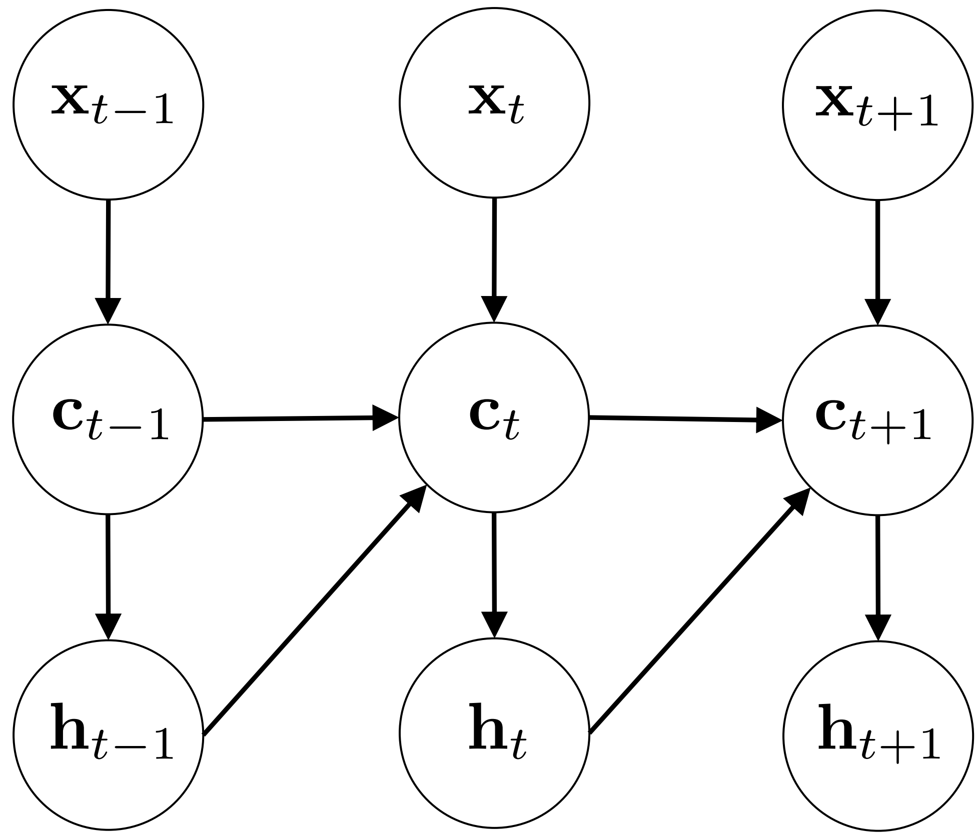
$$\mathbf{c}_t = \mathbf{c}_{t-1} + \tanh\left(\mathbf{W}^{(xc)}\mathbf{x}_t + \mathbf{W}^{(hc)}\mathbf{h}_{t-1} + \mathbf{b}^{(c)}\right)$$

Experiment: text classification

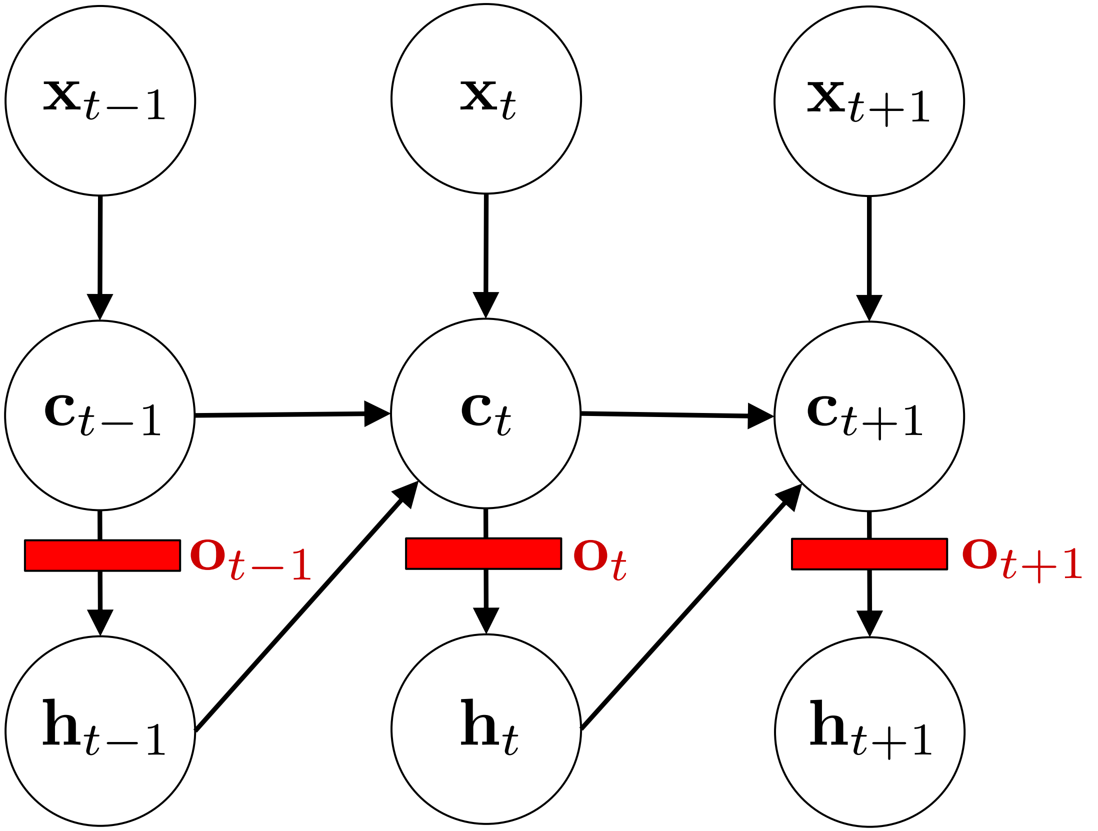
- Stanford Sentiment Treebank
  - binary classification (positive/negative)
- 25-dim word vectors
- 50-dim cell/hidden vectors
- classification layer on **final** hidden vector
- AdaGrad, 10 epochs, mini-batch size 10
- early stopping on dev set

accuracy
80.6

Adding Output Gates

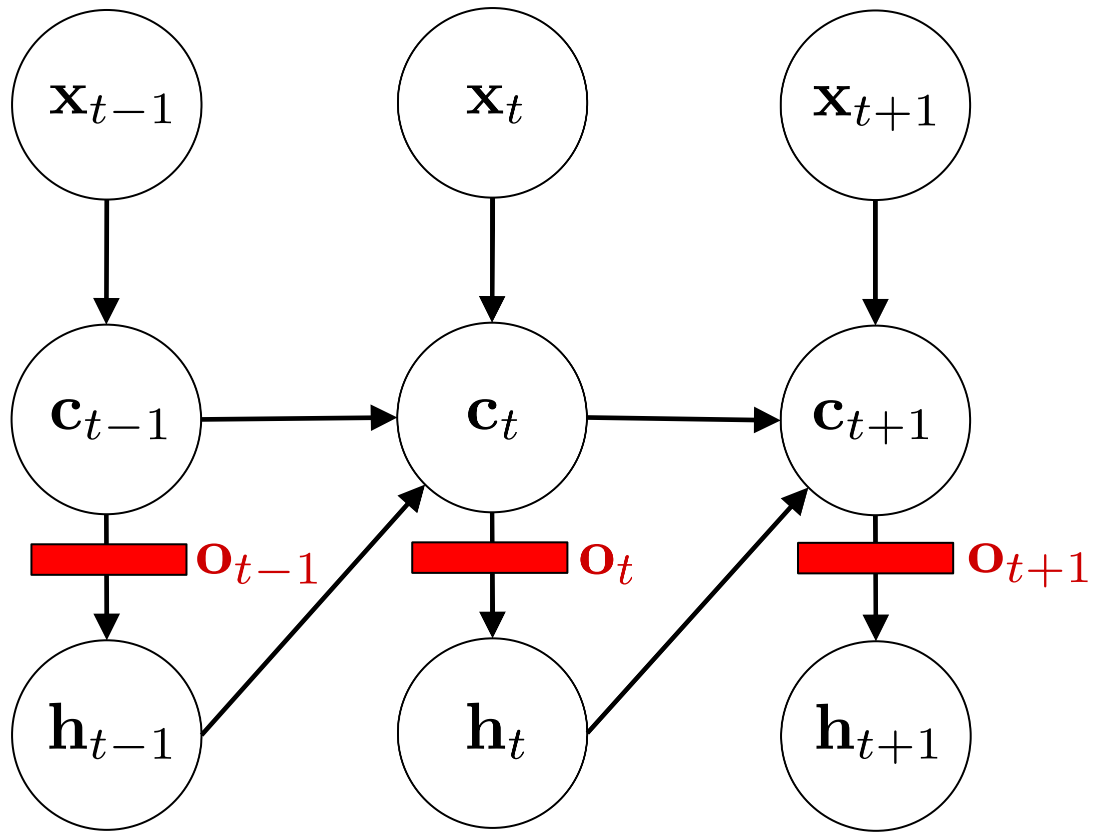


Adding Output Gates



Adding Output Gates

$$\mathbf{h}_t = \tanh(\mathbf{c}_t)$$

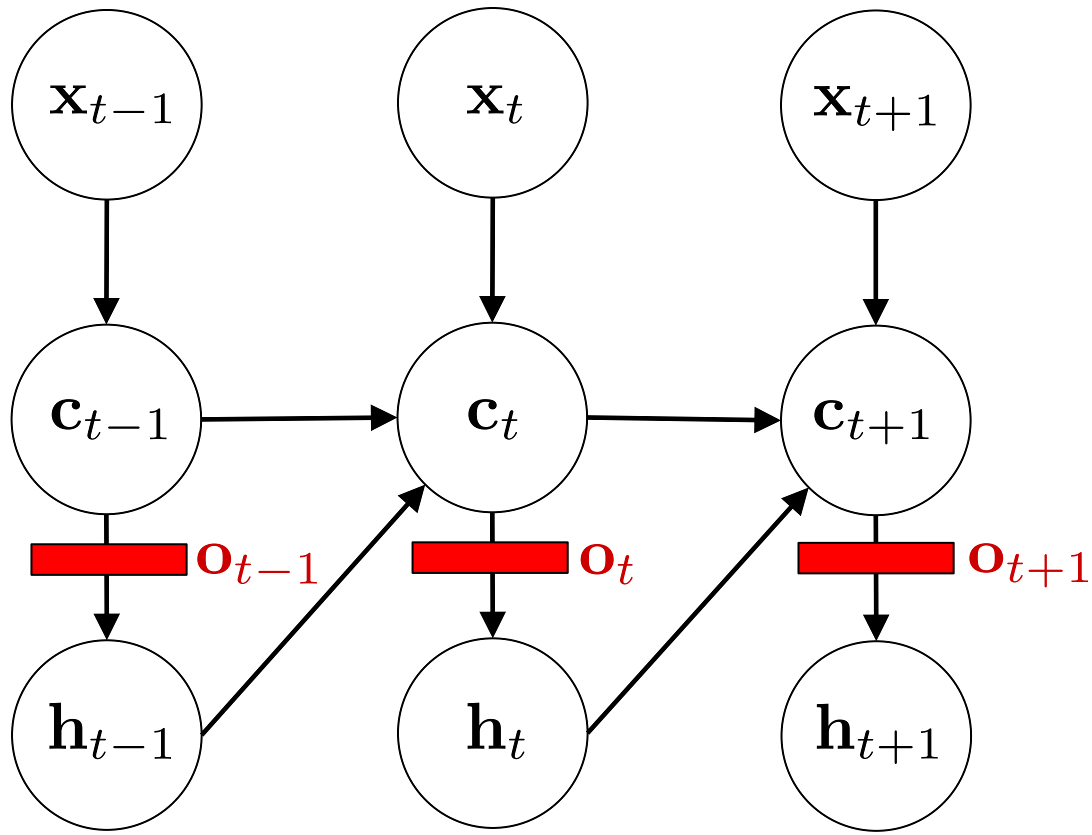


Adding Output Gates

$$\mathbf{h}_t = \tanh(\mathbf{c}_t)$$

$$\Downarrow$$

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

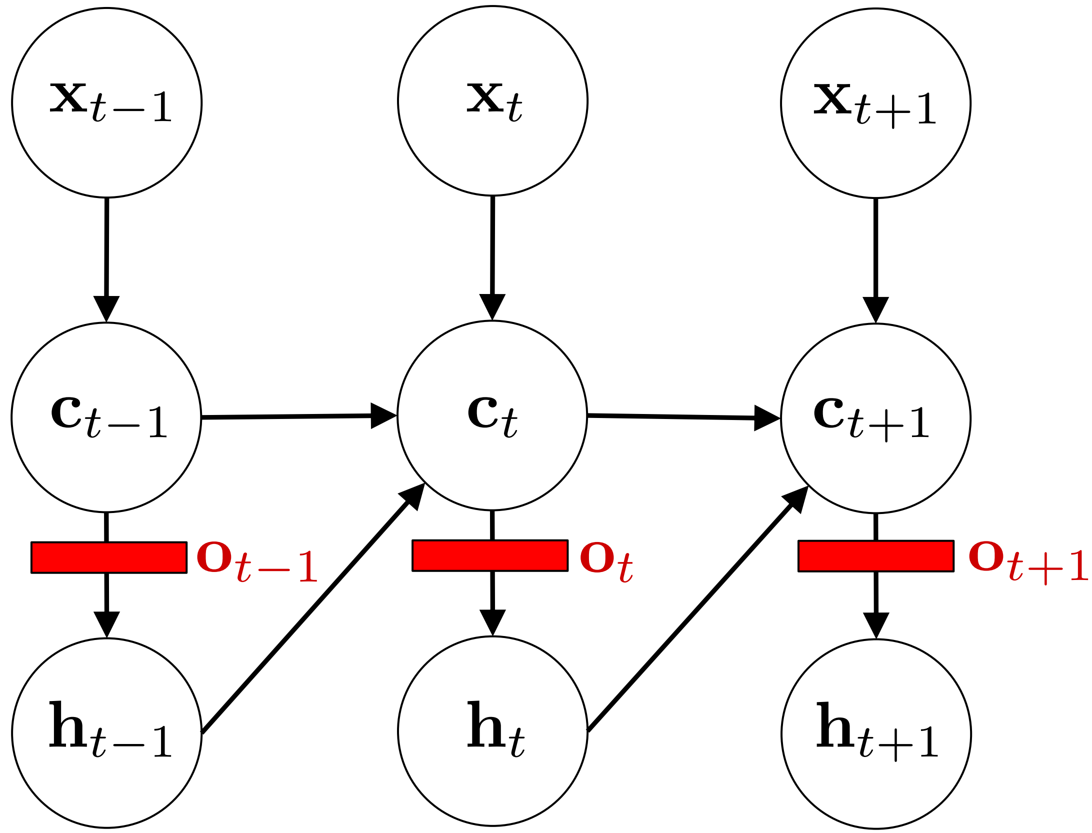


Adding Output Gates

$$\mathbf{h}_t = \tanh(\mathbf{c}_t)$$

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

this is pointwise multiplication!
$\mathbf{o}_t$ is a vector

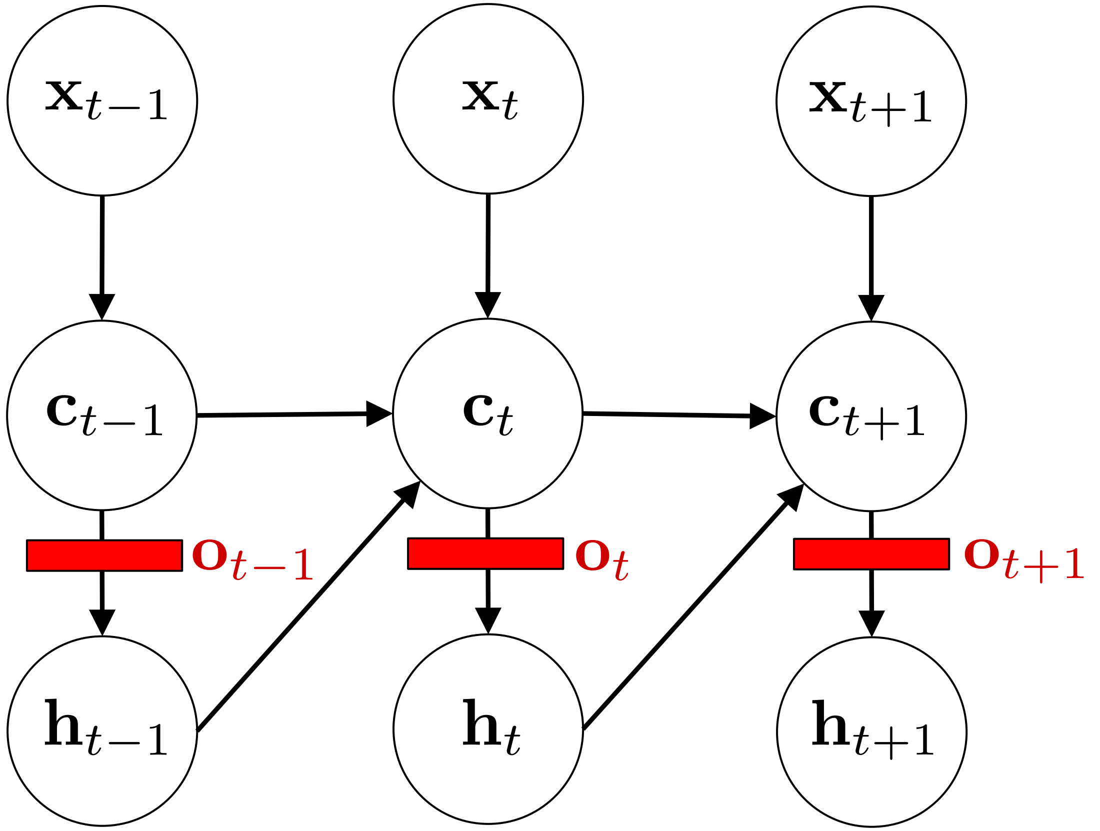


Adding Output Gates

$$\mathbf{h}_t = \tanh(\mathbf{c}_t)$$

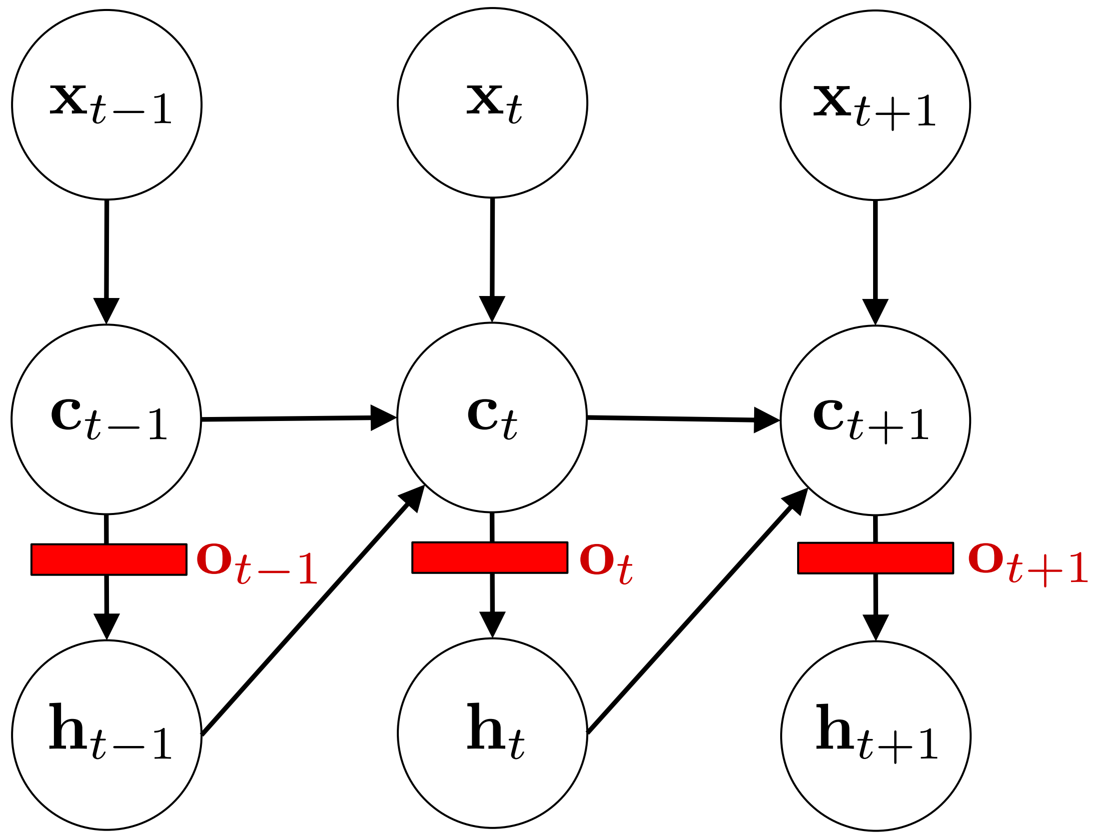
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

output gate affects how much "information" is transmitted from cell vector to hidden vector



Adding Output Gates

$$\mathbf{o}_t = \sigma\left(\mathbf{W}^{(xo)}\mathbf{x}_t + \mathbf{W}^{(ho)}\mathbf{h}_{t-1} + \mathbf{W}^{(co)}\mathbf{c}_t + \mathbf{b}^{(o)}\right)$$



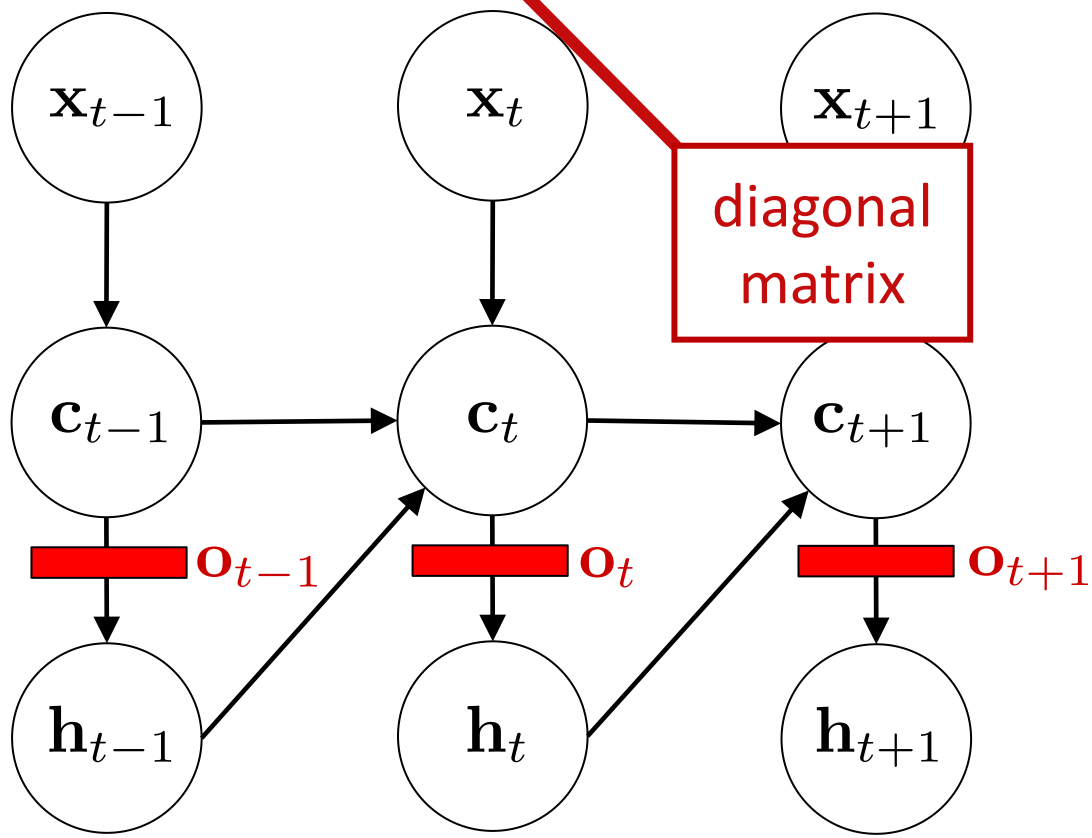
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

Adding Output Gates

$$\mathbf{o}_t = \sigma\left(\mathbf{W}^{(xo)}\mathbf{x}_t + \mathbf{W}^{(ho)}\mathbf{h}_{t-1} + \mathbf{W}^{(co)}\mathbf{c}_t + \mathbf{b}^{(o)}\right)$$

logistic sigmoid, so output ranges from 0 to 1

diagonal matrix

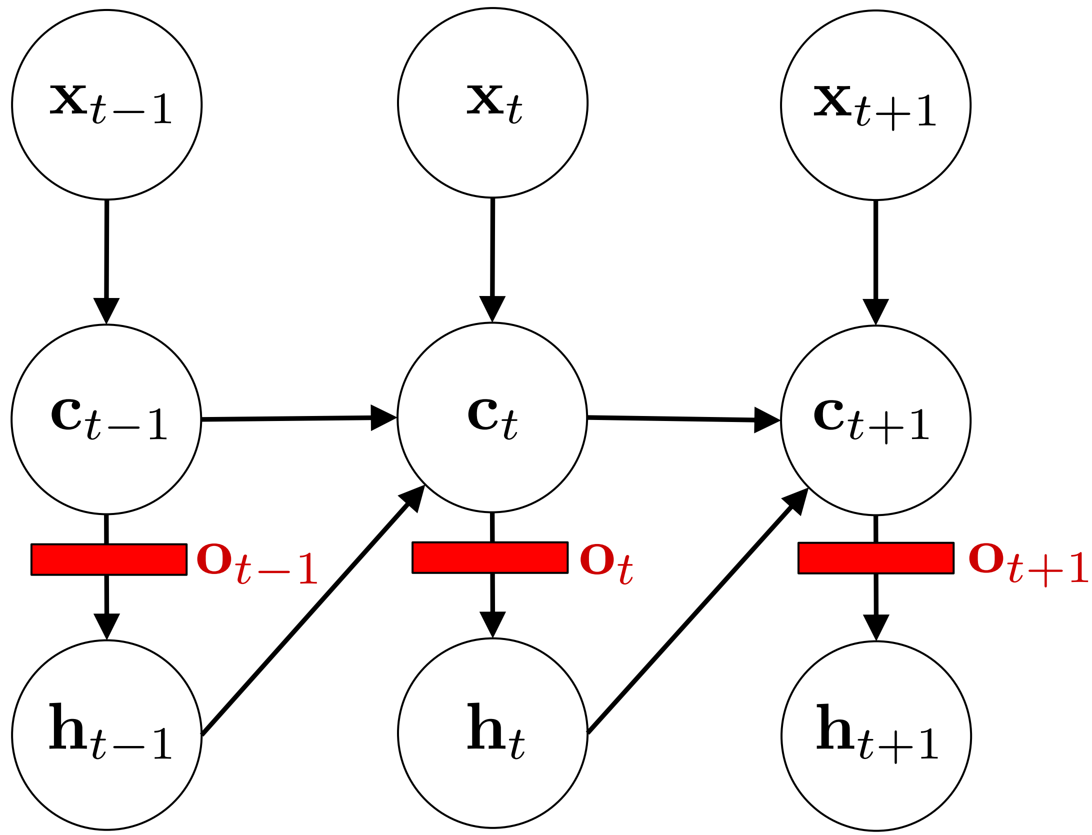


$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

Adding Output Gates

$$\mathbf{o}_t = \sigma\left(\mathbf{W}^{(xo)}\mathbf{x}_t + \mathbf{W}^{(ho)}\mathbf{h}_{t-1} + \mathbf{W}^{(co)}\mathbf{c}_t + \mathbf{b}^{(o)}\right)$$

output gate is a function of current observation, previous hidden vector, and current cell vector

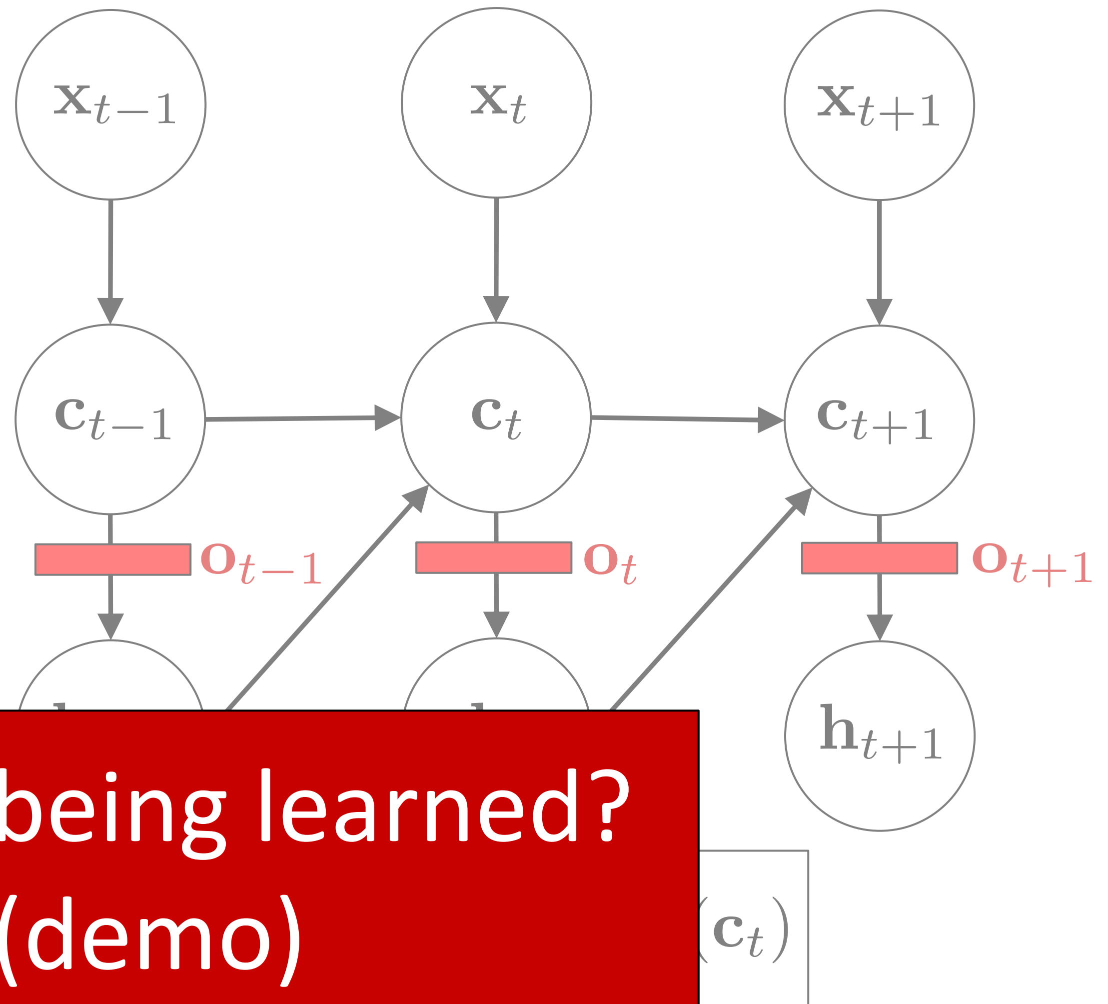


$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

Adding Output Gates

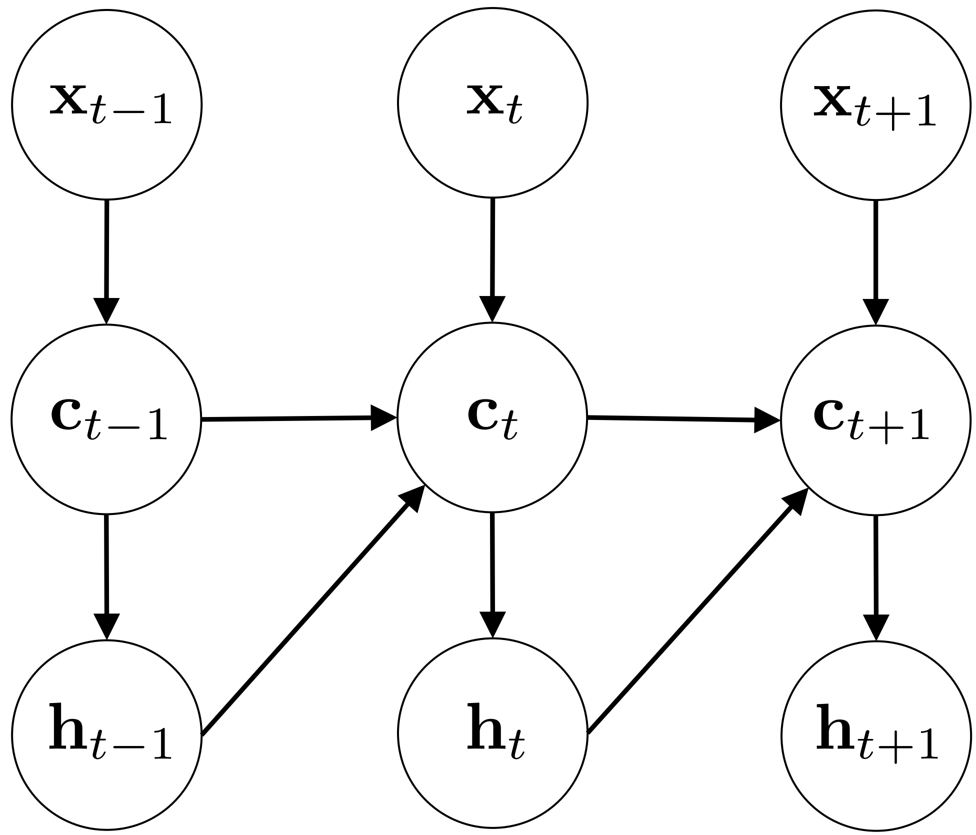
$$\mathbf{o}_t = \sigma\left(\mathbf{W}^{(xo)}\mathbf{x}_t + \mathbf{W}^{(ho)}\mathbf{h}_{t-1} + \mathbf{W}^{(co)}\mathbf{c}_t + \mathbf{b}^{(o)}\right)$$

	acc.
gateless	80.6
output gates	81.9

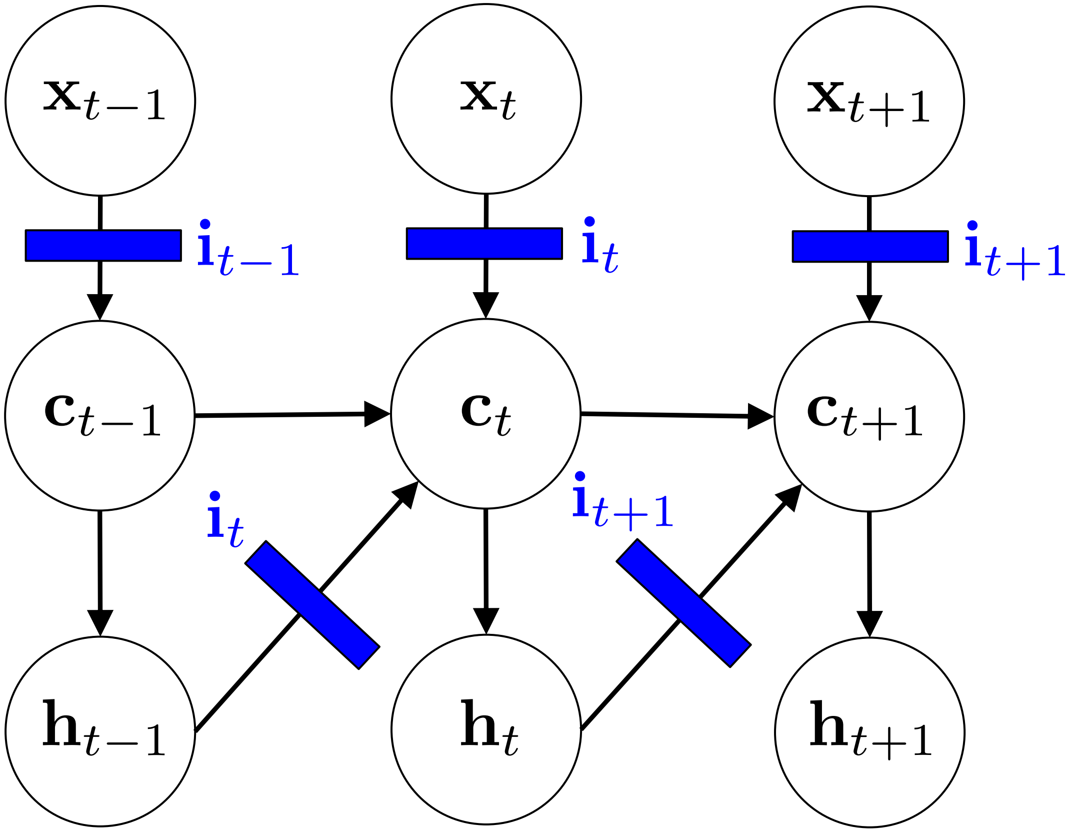


What's being learned?
(demo)

Adding Input Gates



Adding Input Gates

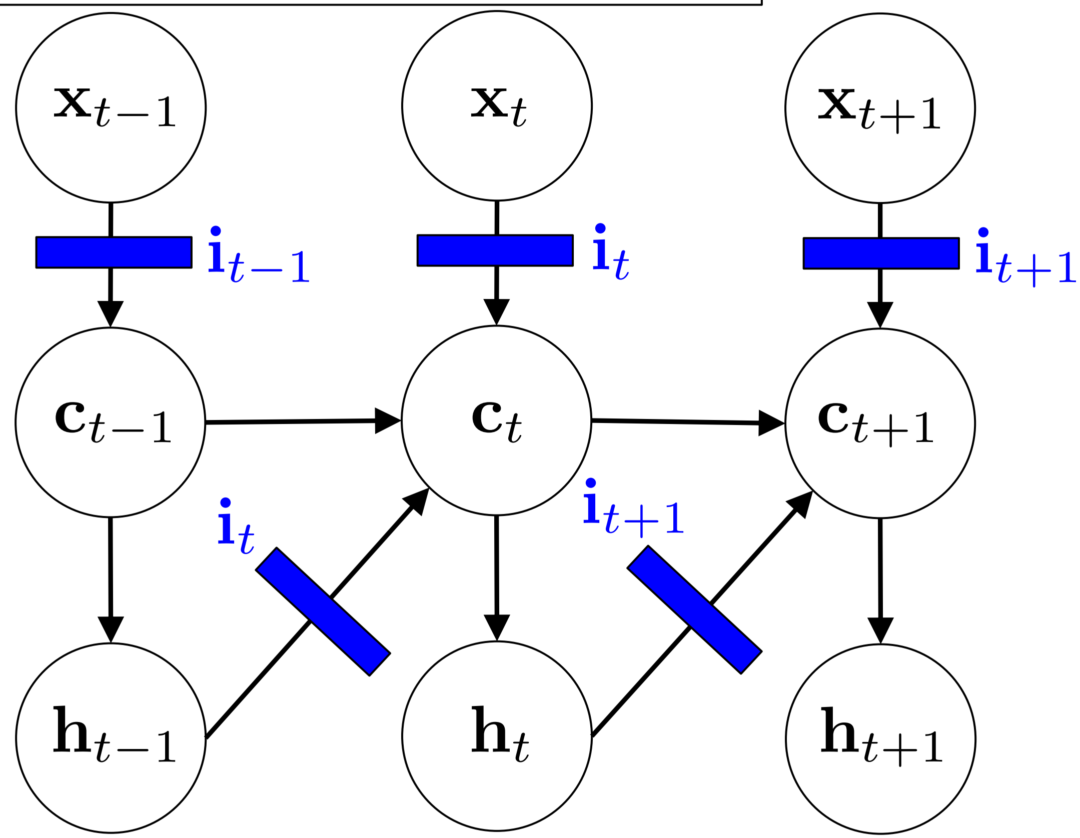


$$\mathbf{c}_t = \mathbf{c}_{t-1} + \tanh\left(\mathbf{W}^{(xc)}\mathbf{x}_t + \mathbf{W}^{(hc)}\mathbf{h}_{t-1} + \mathbf{b}^{(c)}\right)$$

⇩

$$\mathbf{c}_t = \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tanh\left(\mathbf{W}^{(xc)}\mathbf{x}_t + \mathbf{W}^{(hc)}\mathbf{h}_{t-1} + \mathbf{b}^{(c)}\right)$$

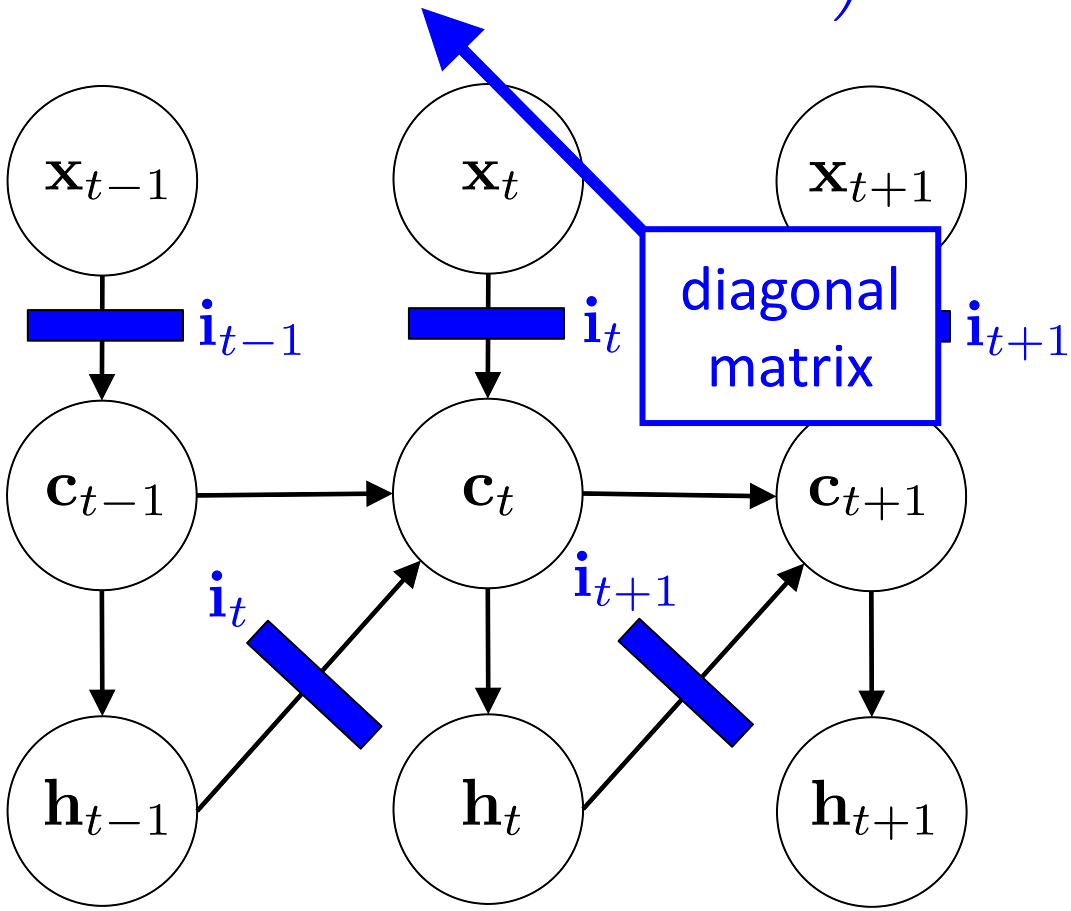
input gate controls how much cell is affected by current observation and previous hidden vector



Input Gates

$$\mathbf{i}_t = \sigma\left(\mathbf{W}^{(xi)}\mathbf{x}_t + \mathbf{W}^{(hi)}\mathbf{h}_{t-1} + \mathbf{W}^{(ci)}\mathbf{c}_{t-1} + \mathbf{b}^{(i)}\right)$$

input gate is a function of current observation, previous hidden vector, and previous cell vector



Input Gates

$$\mathbf{i}_t = \sigma\left(\mathbf{W}^{(xi)}\mathbf{x}_t + \mathbf{W}^{(hi)}\mathbf{h}_{t-1} + \mathbf{W}^{(ci)}\mathbf{c}_{t-1} + \mathbf{b}^{(i)}\right)$$

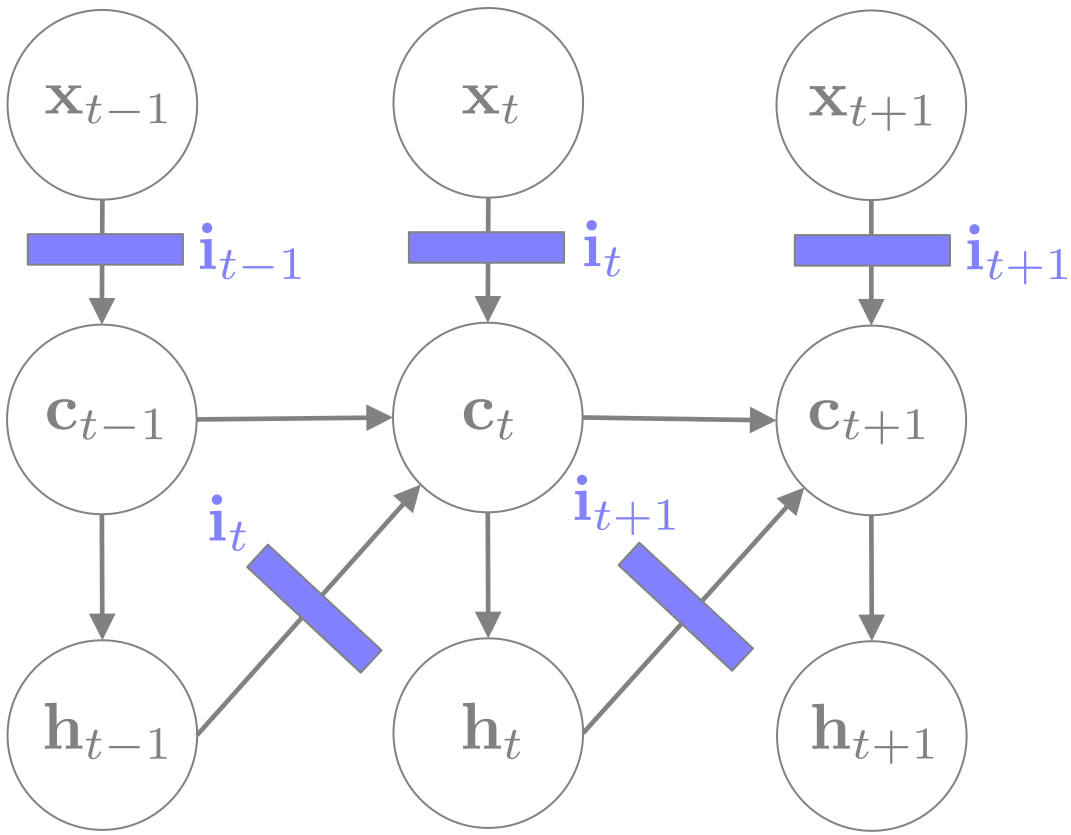
difference

Output Gates

$$\mathbf{o}_t = \sigma\left(\mathbf{W}^{(xo)}\mathbf{x}_t + \mathbf{W}^{(ho)}\mathbf{h}_{t-1} + \mathbf{W}^{(co)}\mathbf{c}_t + \mathbf{b}^{(o)}\right)$$

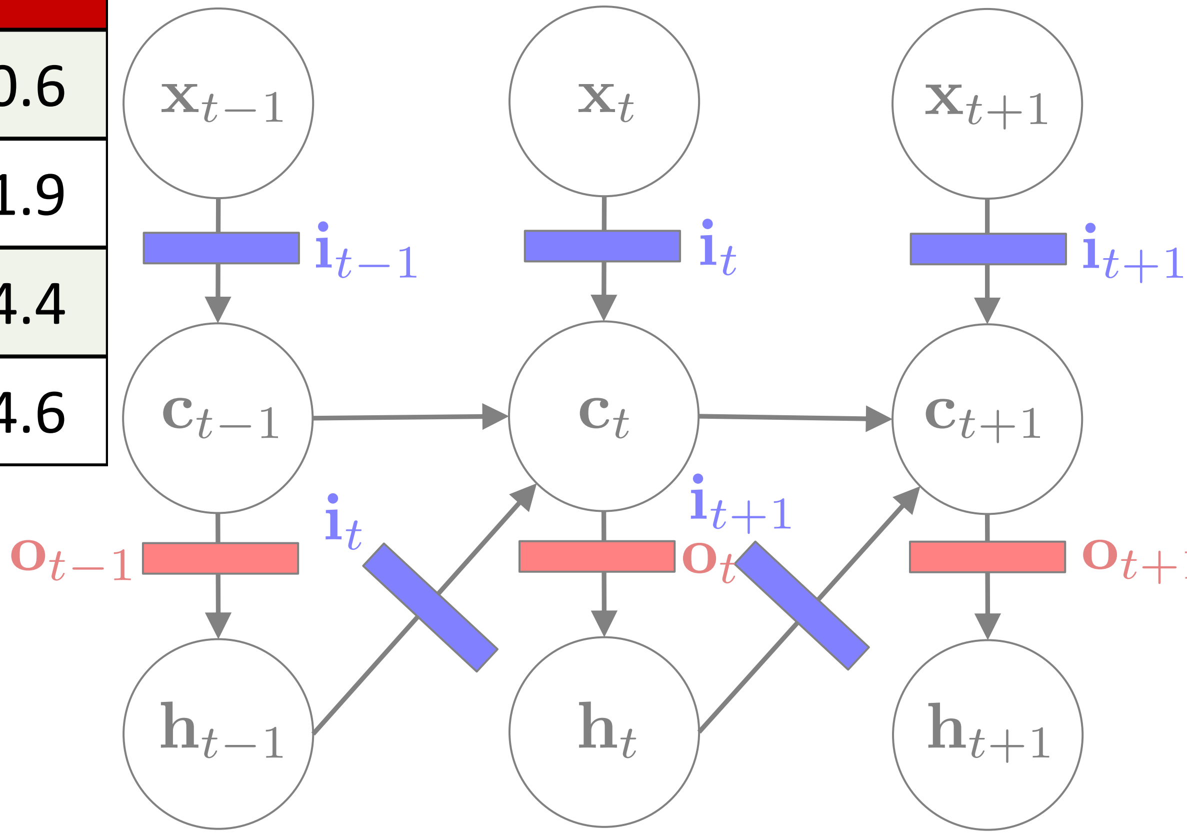
Input Gates

	acc.
gateless	80.6
output gates	81.9
input gates	84.4

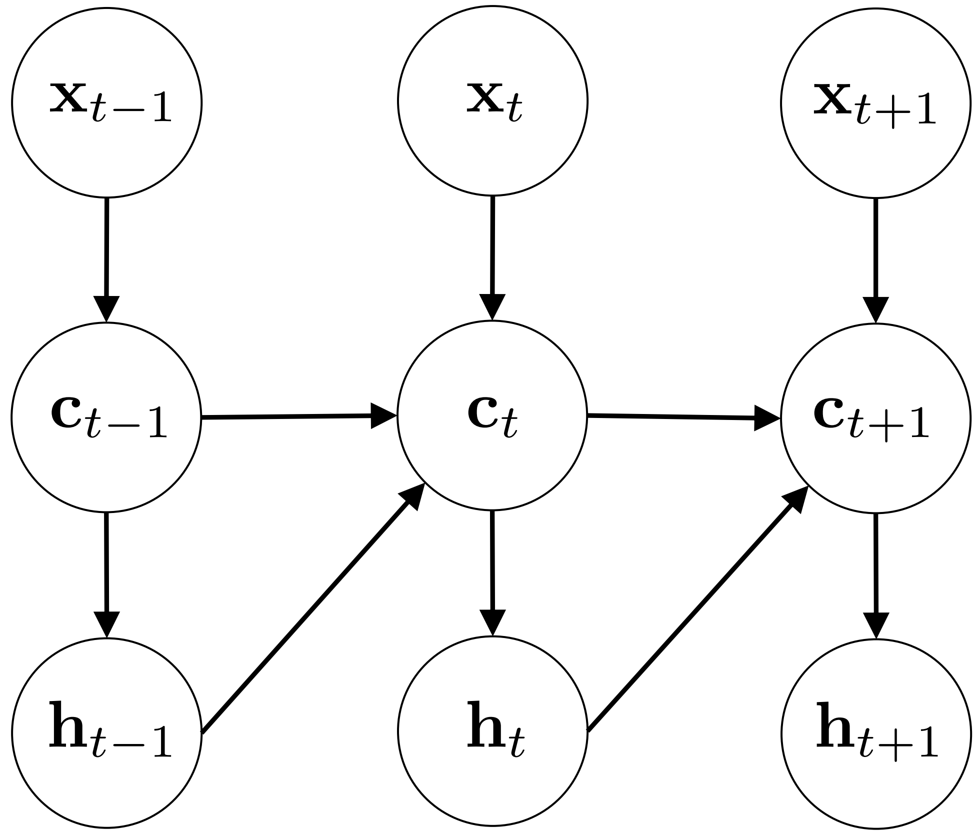


Input and Output Gates

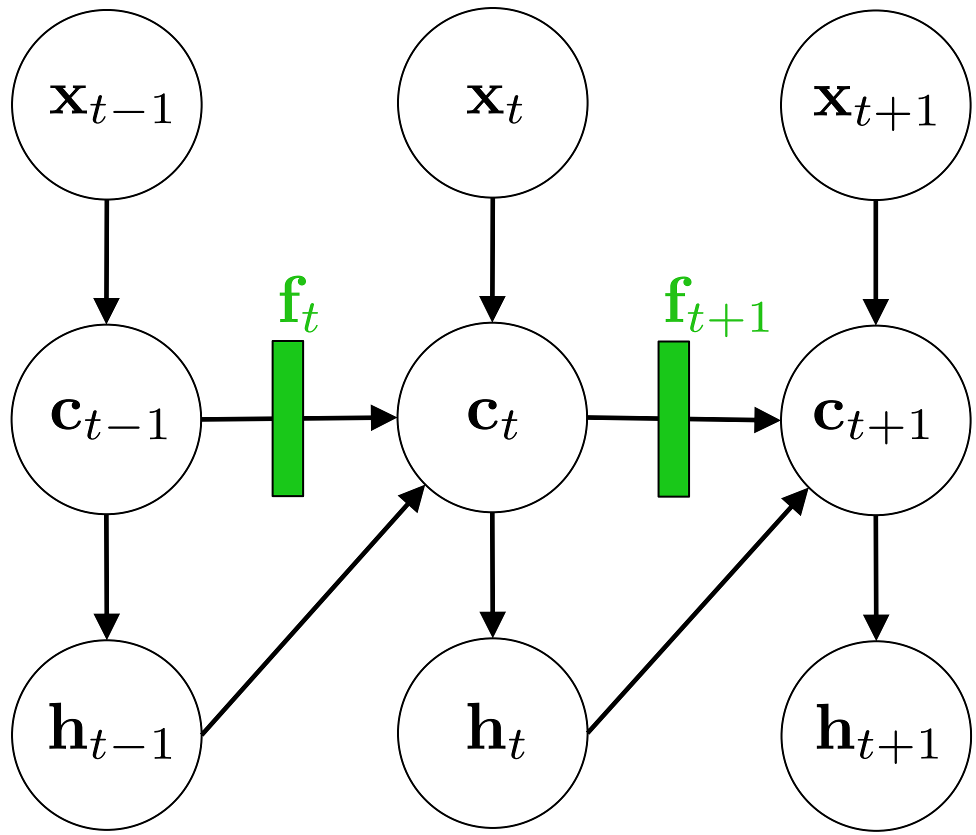
	acc.
gateless	80.6
output gates	81.9
input gates	84.4
input & output gates	84.6



Adding Forget Gates



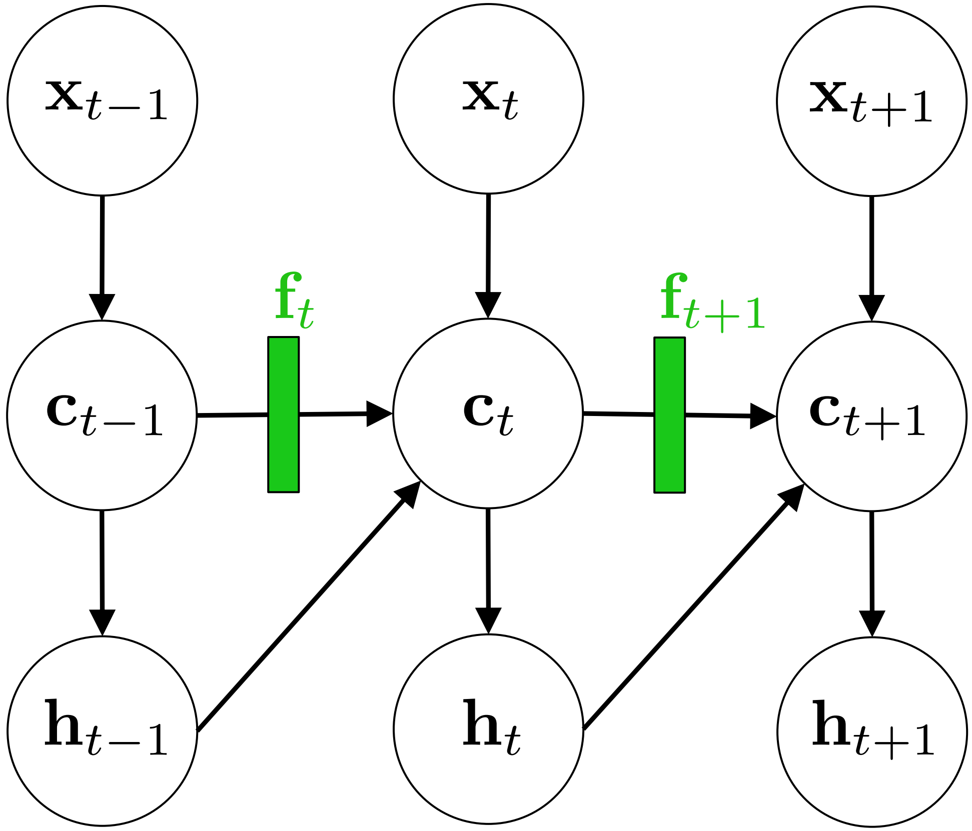
Adding Forget Gates



Adding Forget Gates

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \tanh\left(\mathbf{W}^{(xc)}\mathbf{x}_t + \mathbf{W}^{(hc)}\mathbf{h}_{t-1} + \mathbf{b}^{(c)}\right)$$

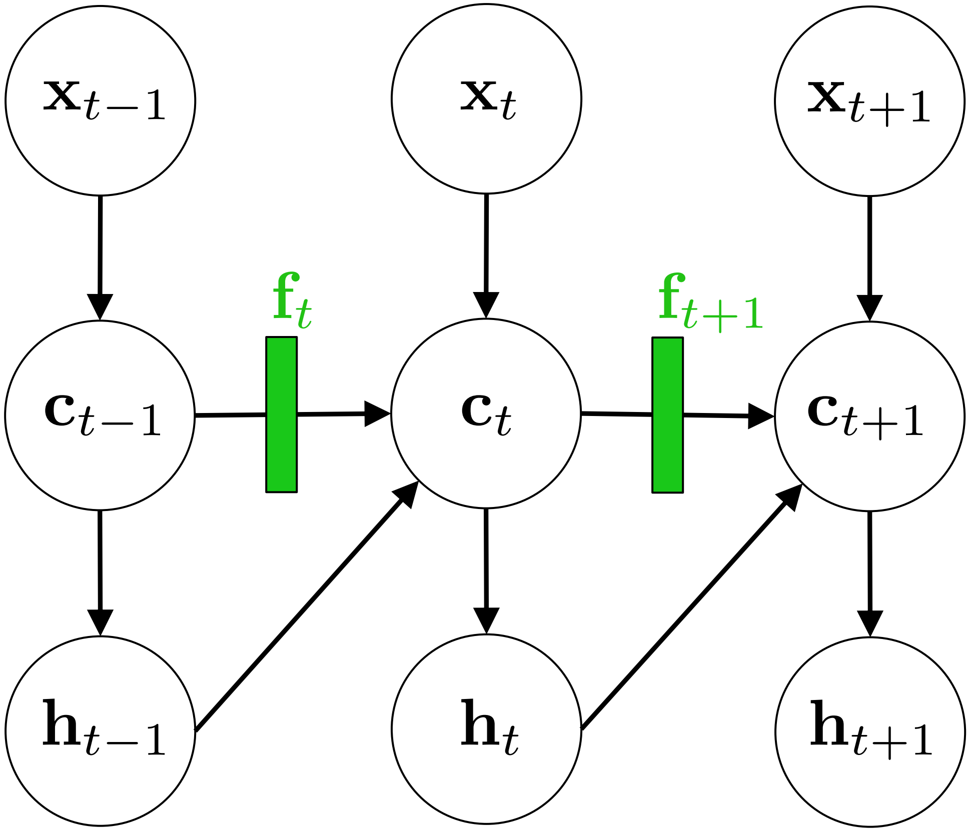
forget gate controls how much "information" is kept from the previous cell vector



Adding Forget Gates

$$\mathbf{f}_t = \sigma\left(\mathbf{W}^{(xf)}\mathbf{x}_t + \mathbf{W}^{(hf)}\mathbf{h}_{t-1} + \mathbf{W}^{(cf)}\mathbf{c}_{t-1} + \mathbf{b}^{(f)}\right)$$

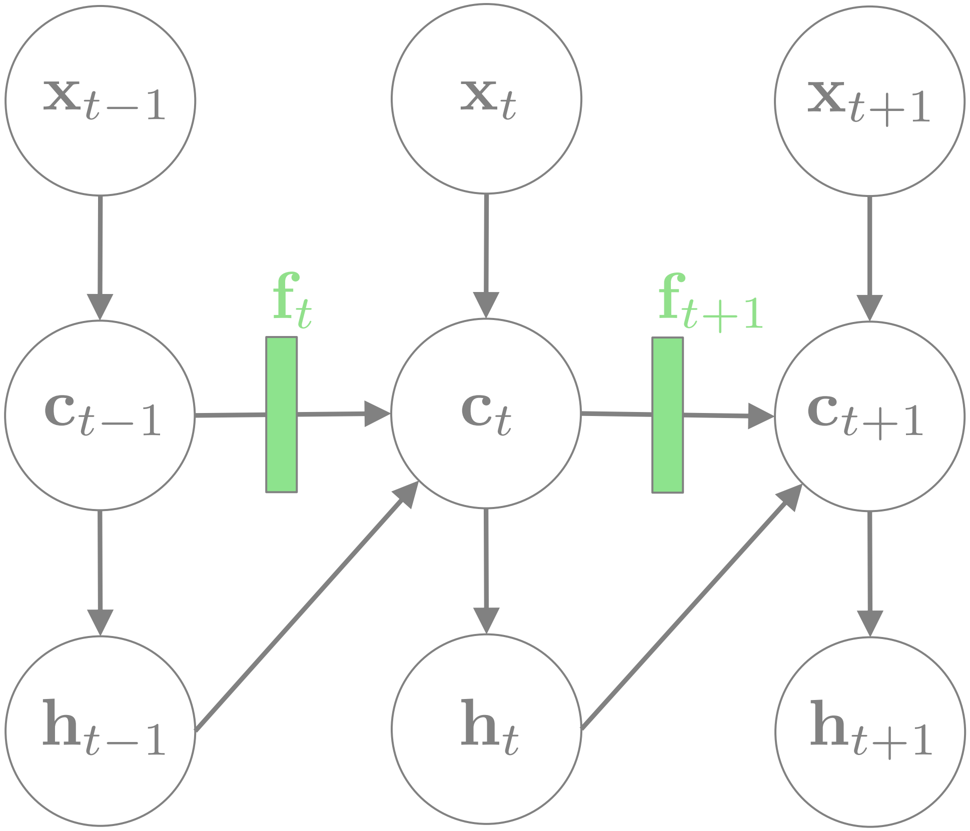
forget gate depends on current observation, previous hidden vector, and previous cell vector



Adding Forget Gates

$$\mathbf{f}_t = \sigma\left(\mathbf{W}^{(xf)}\mathbf{x}_t + \mathbf{W}^{(hf)}\mathbf{h}_{t-1} + \mathbf{W}^{(cf)}\mathbf{c}_{t-1} + \mathbf{b}^{(f)}\right)$$

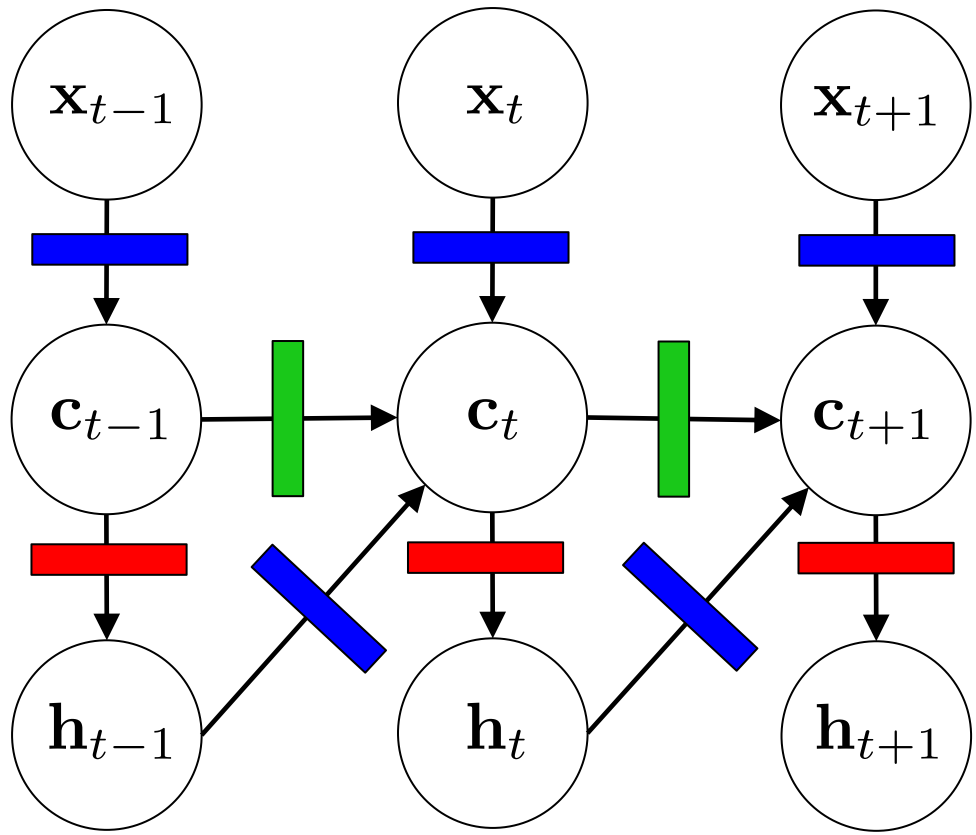
	acc.
gateless	80.6
output gates	81.9
input gates	84.4
forget gates	82.1



All Gates

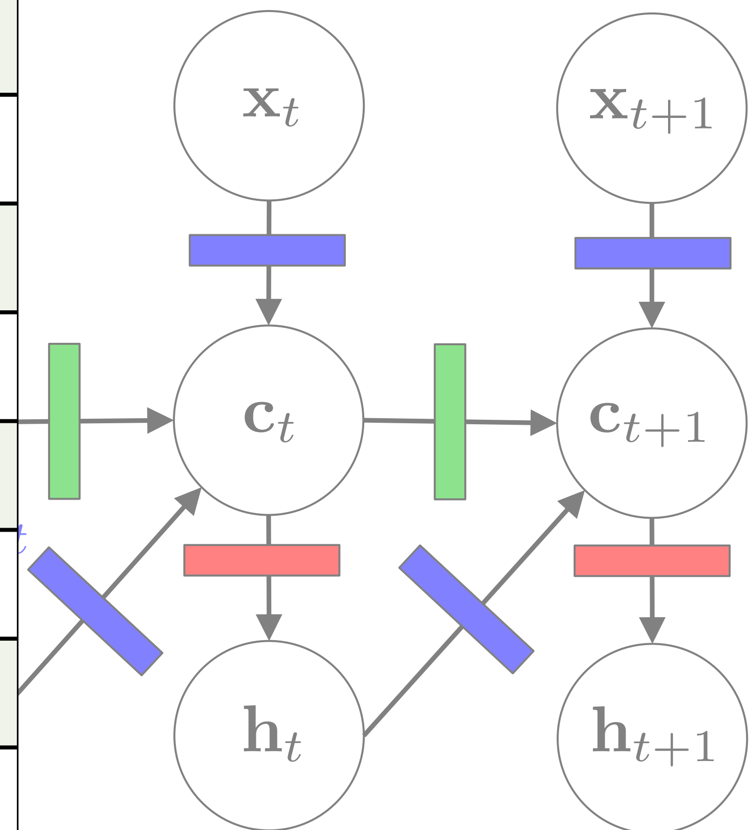
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tanh\left(\mathbf{W}^{(xc)}\mathbf{x}_t + \mathbf{W}^{(hc)}\mathbf{h}_{t-1} + \mathbf{b}^{(c)}\right)$$

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

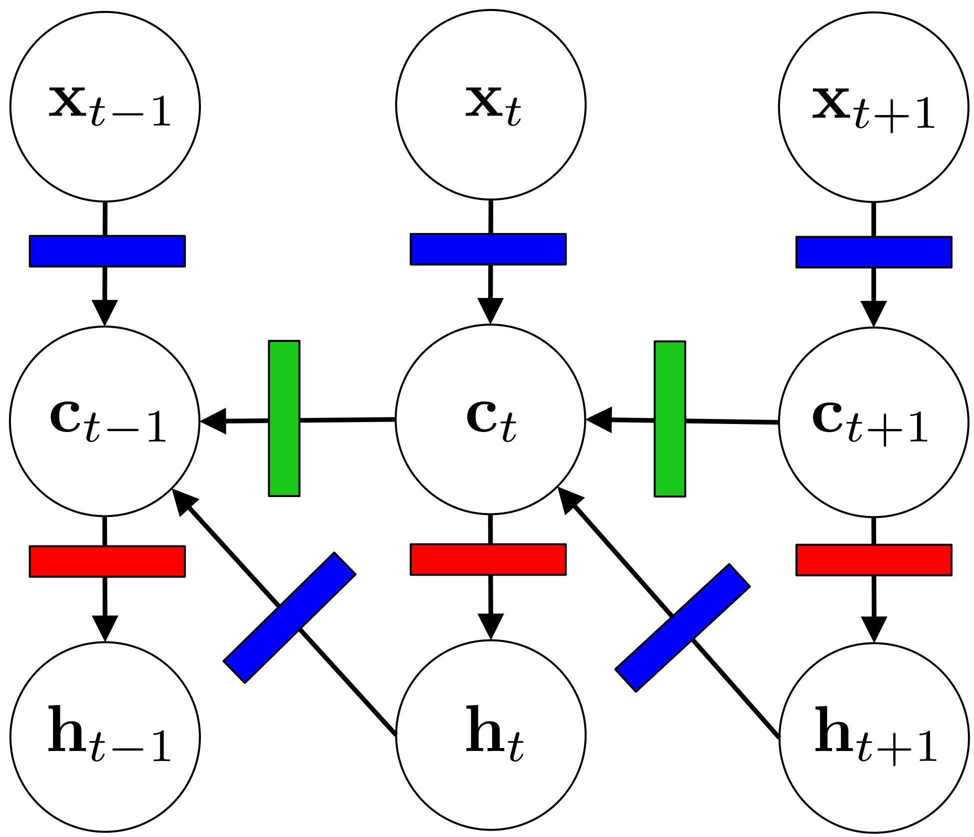


All Gates

	acc.
gateless	80.6
output gates	81.9
input gates	84.4
input & output gates	84.6
forget gates	82.1
input & forget gates	84.1
forget & output gates	82.6
input, forget, output gates	**85.3**

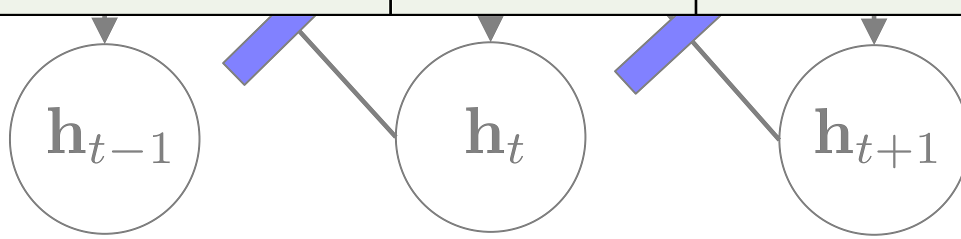


Backward LSTMs



Backward LSTMs

	forward	backward
gateless	80.6	80.3
output gates	81.9	83.7
input gates	84.4	82.9
forget gates	82.1	83.4
input, forget, output gates	85.3	85.9



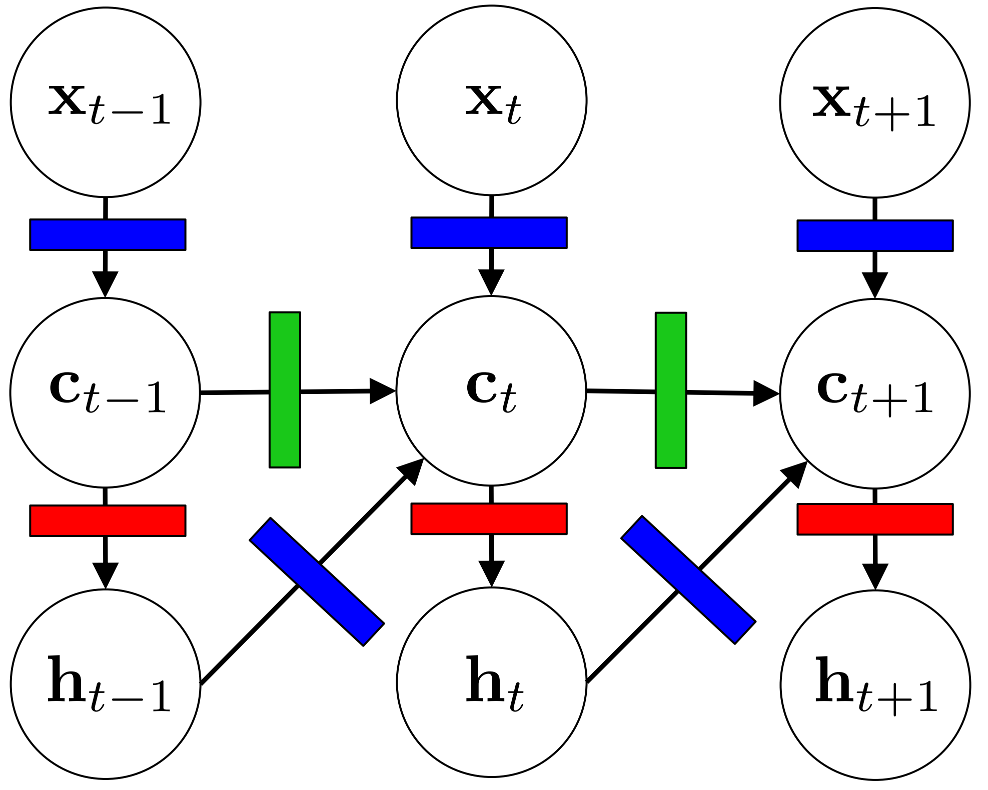
Bidirectional LSTMs

bidirectional:

if shallow, just use forward and backward LSTMs in parallel, concatenate final two hidden vectors, feed to softmax

	forward	backward	bidirectional
gateless	80.6	80.3	81.5
output gates	81.9	83.7	82.6
input gates	84.4	82.9	83.9
forget gates	82.1	83.4	83.1
input, forget, output gates	85.3	85.9	85.1

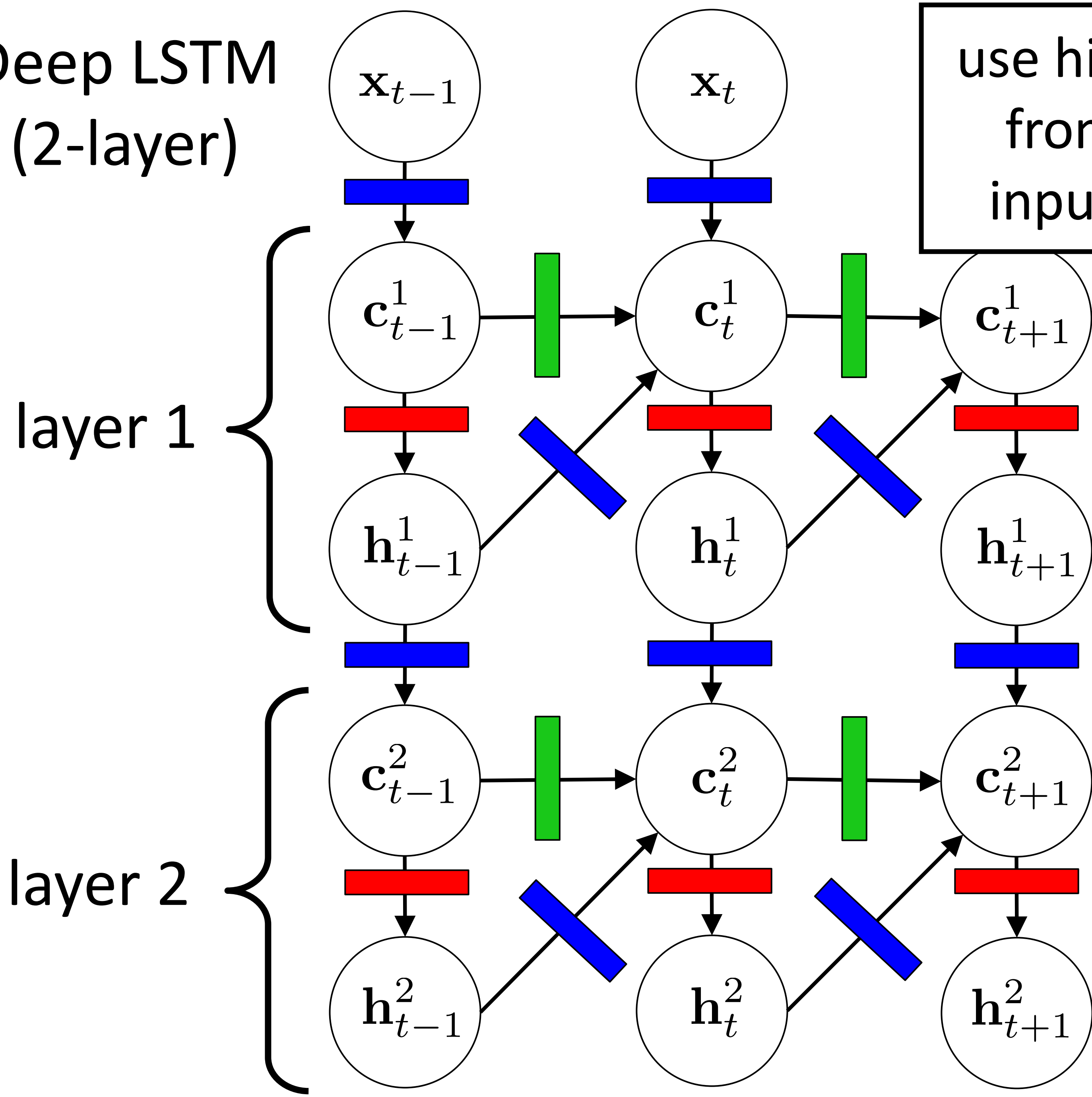
LSTM



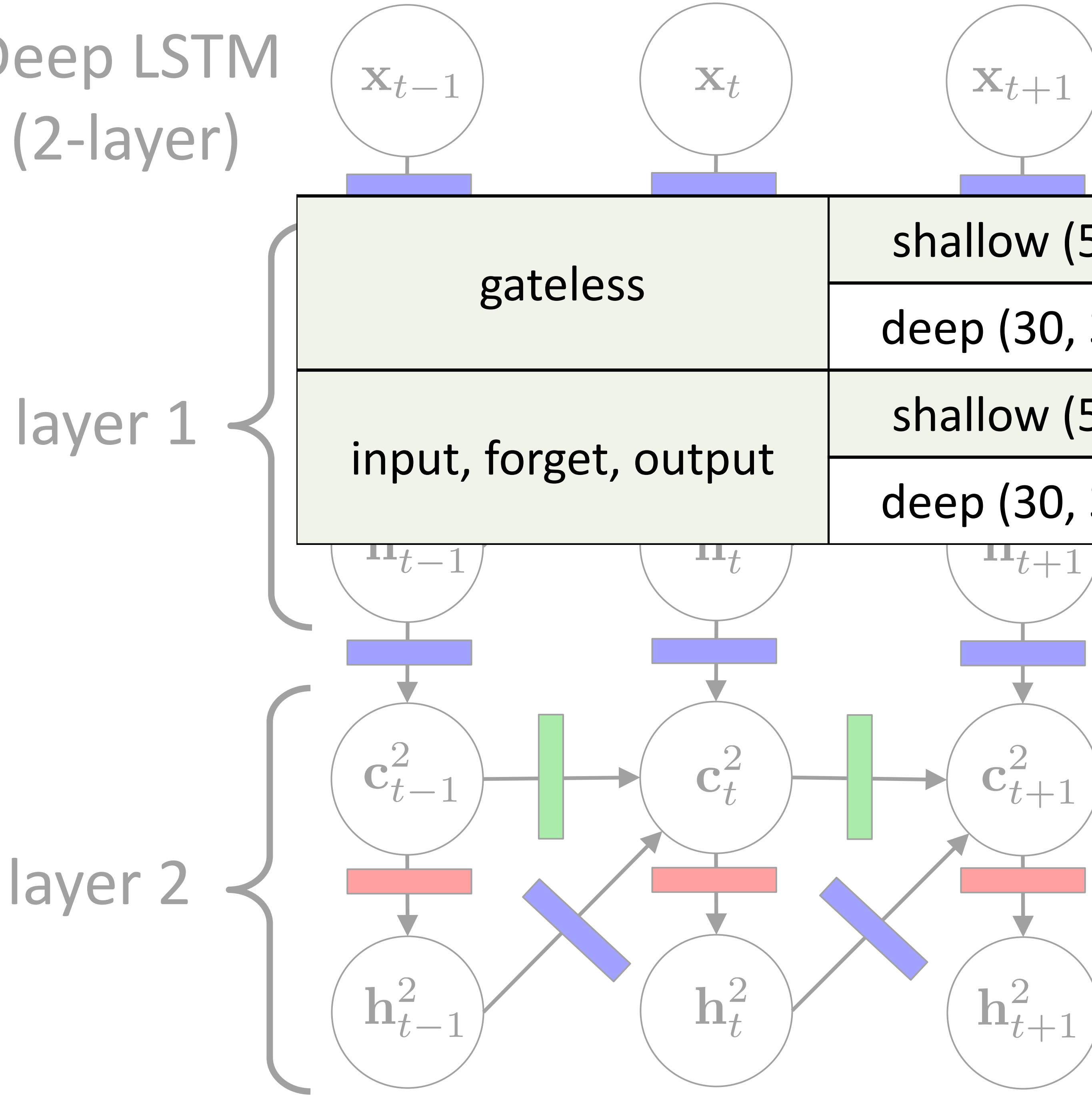
$\mathbf{x}_{t-1}$
$\mathbf{x}_t$
$\mathbf{x}_{t+1}$
$\mathbf{c}_{t-1}$
$\mathbf{c}_t$
$\mathbf{c}_{t+1}$
$\mathbf{h}_{t-1}$
$\mathbf{h}_t$
$\mathbf{h}_{t+1}$

Deep LSTM (2-layer)

use hidden vectors from layer 1 as inputs to layer 2



Deep LSTM (2-layer)



		acc.
gateless	shallow (50)	80.6
	deep (30, 30)	80.8
input, forget, output	shallow (50)	85.3
	deep (30, 30)	~85

Deep Bidirectional LSTMs

concatenate hidden vectors of forward & backward LSTMs, connect each entry to forward and backward hidden vectors in next layer

Gated Recurrent Units (GRU)

- alternative to LSTMs, fewer parameters, generally works pretty well

Gated Recurrent Units (GRU)

- alternative to LSTMs, fewer parameters, generally works pretty well
- uses "reset" and "update" gates instead of LSTM gates:

$$\mathbf{h}_t = (\mathbf{1} - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tanh\left(\mathbf{W}\mathbf{x}_t + \mathbf{U}(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}\right)$$

update gate

reset gate

Recursive Neural Networks for NLP

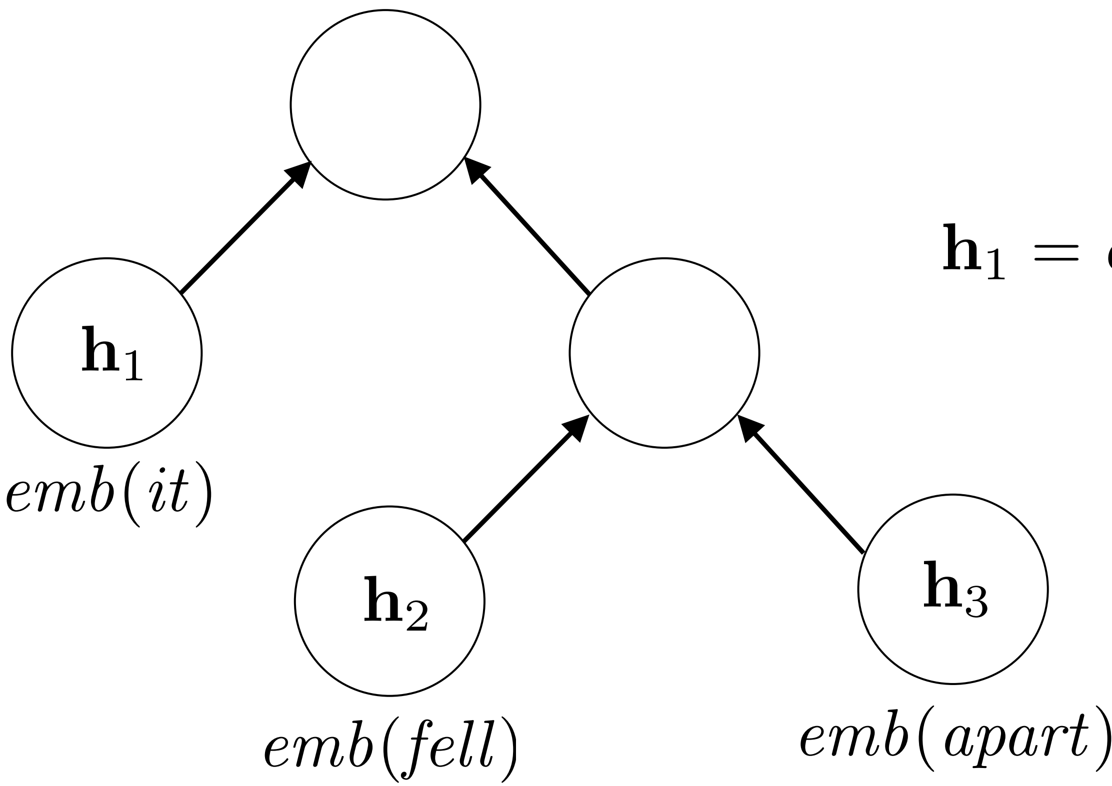
$x =$ *it fell apart*

- run a syntactic parser on the sentence
- construct vector recursively at each split point:

Recursive Neural Networks for NLP

$x =$ *it fell apart*

- run a syntactic parser on the sentence
- construct vector recursively at each split point:

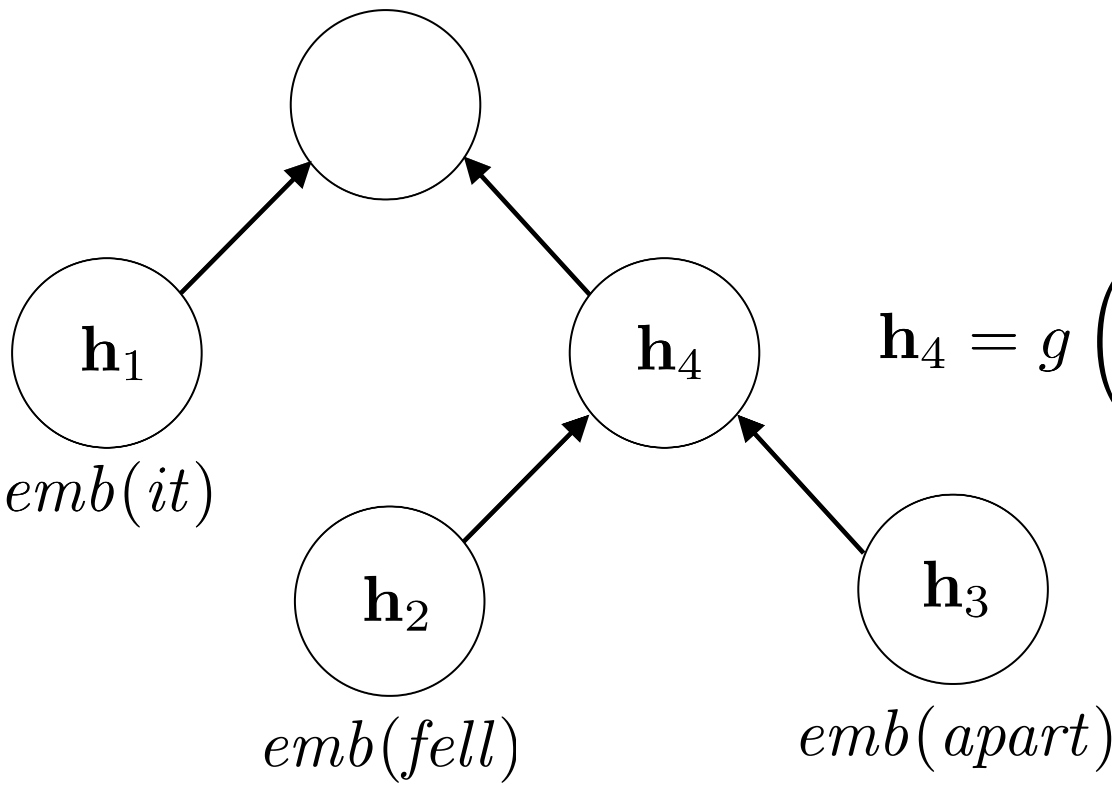


$$\mathbf{h}_1 = emb(it)$$

Recursive Neural Networks for NLP

$\boldsymbol{x} =$ *it fell apart*

- run a syntactic parser on the sentence
- construct vector recursively at each split point:

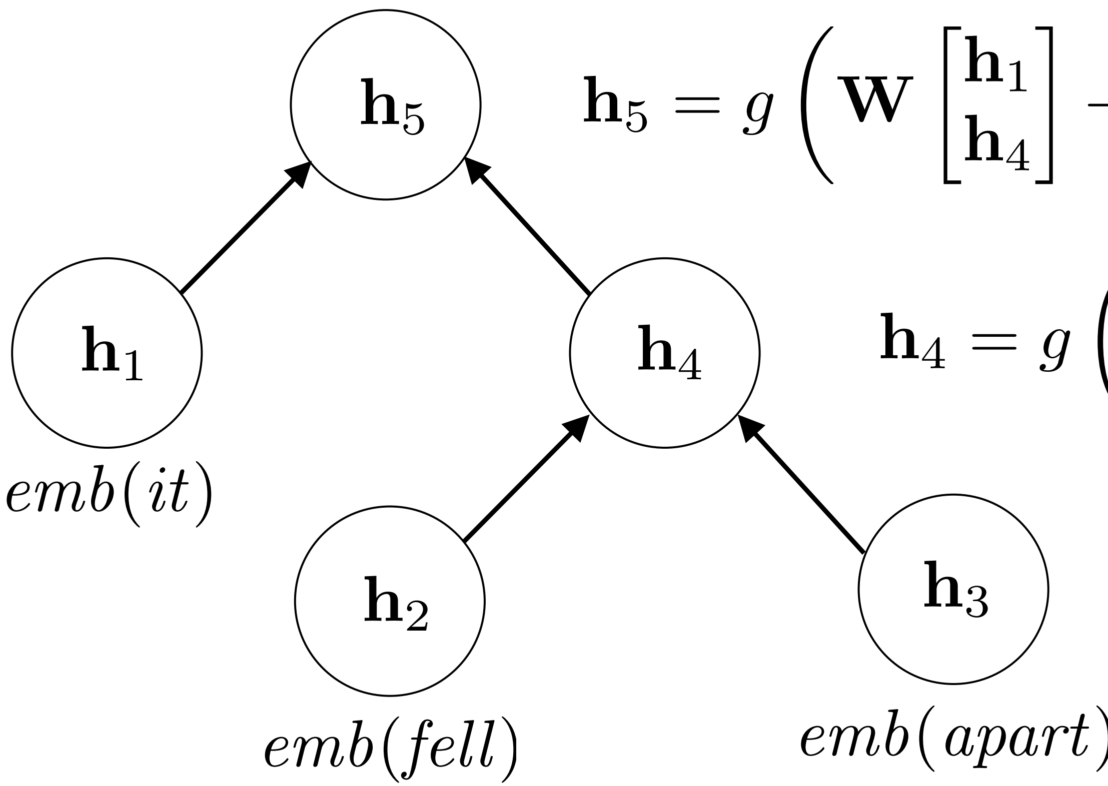


$$\mathbf{h}_4 = g\left(\mathbf{W}\begin{bmatrix}\mathbf{h}_2 \\ \mathbf{h}_3\end{bmatrix} + \mathbf{b}\right)$$

Recursive Neural Networks for NLP

$\boldsymbol{x} =$ *it fell apart*

- run a syntactic parser on the sentence
- construct vector recursively at each split point:

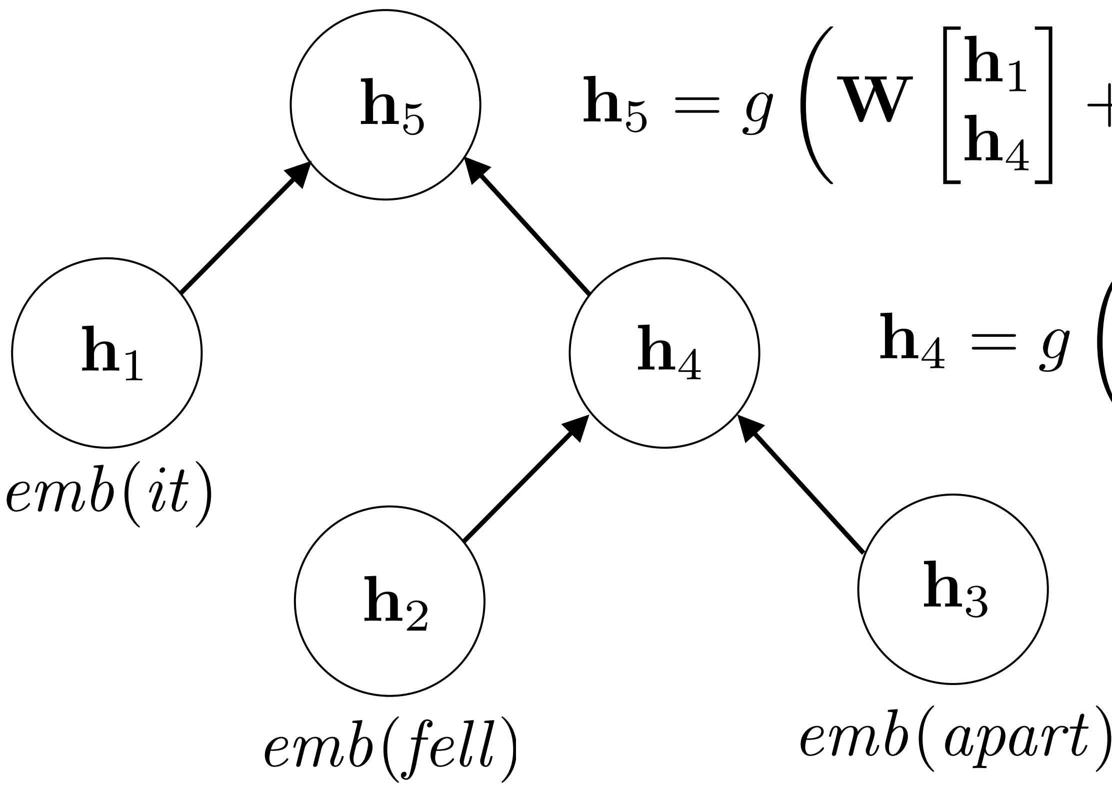


$$\mathbf{h}_5 = g\left(\mathbf{W}\begin{bmatrix}\mathbf{h}_1\\ \mathbf{h}_4\end{bmatrix} + \mathbf{b}\right)$$

$$\mathbf{h}_4 = g\left(\mathbf{W}\begin{bmatrix}\mathbf{h}_2\\ \mathbf{h}_3\end{bmatrix} + \mathbf{b}\right)$$

Recursive Neural Networks for NLP

- same parameters used at every split point
- order of children matters (different weights used for left and right child)



$$\mathbf{h}_5 = g\left(\mathbf{W}\begin{bmatrix}\mathbf{h}_1\\ \mathbf{h}_4\end{bmatrix} + \mathbf{b}\right)$$

$$\mathbf{h}_4 = g\left(\mathbf{W}\begin{bmatrix}\mathbf{h}_2\\ \mathbf{h}_3\end{bmatrix} + \mathbf{b}\right)$$

Convolutional Neural Networks

- convolutional neural networks (**convnets** or **CNNs**) use **filters** that are "convolved with" (matched against all positions of) the input
- informally, think of convolution as "perform the same operation everywhere on the input in some systematic order"
- CNNs are often used in NLP to convert a sentence into a feature vector

Filters

- for now, think of a filter as a vector in the word vector space
- the filter matches a particular region of the space
- “match” = “has high dot product with”

Convolution

$\boldsymbol{x} =$ *not that great*

$$\mathbf{x} = [0.4 \ldots 0.9 \;\; 0.2 \ldots 0.7 \;\; 0.3 \ldots 0.6]^\top$$

vector for *not* vector for *that* vector for *great*

consider a single convolutional filter $\mathbf{w} \in \mathbb{R}^d$

Convolution

compute dot product of filter and each word vector:

$\boldsymbol{x} =$ *not that great*

$$\mathbf{x} = \overset{\mathbf{w}}{[0.4 \; ... \; 0.9} \;\; 0.2 \; ... \; 0.7 \;\; 0.3 \; ... \; 0.6]^\top$$

vector for *not* | vector for *that* | vector for *great*

$$c_1 = \mathbf{w} \cdot \mathbf{x}_{1:d}$$

Convolution

compute dot product of filter and each word vector:

$\boldsymbol{x} =$ *not that great*

$\mathbf{w}$

$$\mathbf{x} = [0.4 \ldots 0.9 \;\; 0.2 \ldots 0.7 \;\; 0.3 \ldots 0.6]^\top$$

vector for *not* — vector for *that* — vector for *great*

$$c_1 = \mathbf{w} \cdot \mathbf{x}_{1:d}$$

$$c_2 = \mathbf{w} \cdot \mathbf{x}_{d+1:2d}$$

Convolution

compute dot product of filter and each word vector:

$\boldsymbol{x} =$ *not that great*

$$\mathbf{x} = [\underbrace{0.4 \ldots 0.9}_{\text{vector for } not} \; \underbrace{0.2 \ldots 0.7}_{\text{vector for } that} \; \underbrace{0.3 \overset{\mathbf{w}}{\ldots} 0.6}_{\text{vector for } great}]^\top$$

$$c_1 = \mathbf{w} \cdot \mathbf{x}_{1:d}$$

$$c_2 = \mathbf{w} \cdot \mathbf{x}_{d+1:2d}$$

$$c_3 = \mathbf{w} \cdot \mathbf{x}_{2d+1:3d}$$

Convolution

$\boldsymbol{x} =$ *not that great*

$$\mathbf{x} = [\underbrace{0.4 \;\ldots\; 0.9}_{\text{vector for } \textit{not}} \;\; \underbrace{0.2 \;\ldots\; 0.7}_{\text{vector for } \textit{that}} \;\; \underbrace{0.3 \;\ldots\; 0.6}_{\text{vector for } \textit{great}}]^{\top}$$

$$c_1 = \mathbf{w} \cdot \mathbf{x}_{1:d}$$

$$c_2 = \mathbf{w} \cdot \mathbf{x}_{d+1:2d}$$

$$c_3 = \mathbf{w} \cdot \mathbf{x}_{2d+1:3d}$$

Note: it’s common to add a bias *b* and use a nonlinearity *g:*

$$c_1 = g\left(\mathbf{w} \cdot \mathbf{x}_{1:d} + b\right)$$

Convolution

$\boldsymbol{x} =$ *not that great*

$$\mathbf{x} = [0.4 \ldots 0.9 \;\; 0.2 \ldots 0.7 \;\; 0.3 \ldots 0.6]^\top$$

vector for *not* vector for *that* vector for *great*

$$c_1 = \mathbf{w} \cdot \mathbf{x}_{1:d}$$

$$c_2 = \mathbf{w} \cdot \mathbf{x}_{d+1:2d}$$

$$c_3 = \mathbf{w} \cdot \mathbf{x}_{2d+1:3d}$$

$\mathbf{c}$ = "feature map" for this filter,
has an entry for each position in input (in this case, 3 entries)

Pooling

$\boldsymbol{x} =$ *not that great*

how do we convert this into a fixed-length vector?
use **pooling**:
- max-pooling: returns maximum value in $\mathbf{c}$
- average pooling: returns average of values in $\mathbf{c}$

$$c_1 = \mathbf{w} \cdot \mathbf{x}_{1:d}$$

$$c_2 = \mathbf{w} \cdot \mathbf{x}_{d+1:2d}$$

$$c_3 = \mathbf{w} \cdot \mathbf{x}_{2d+1:3d}$$

Pooling

$\boldsymbol{x} =$ *not that great*

how do we convert this into a fixed-length vector?

use **pooling**:

- max-pooling: returns maximum value in $\mathbf{c}$
- average pooling: returns average of values in $\mathbf{c}$

$$c_1 = \mathbf{w} \cdot \mathbf{x}_{1:d}$$

$$c_2 = \mathbf{w} \cdot \mathbf{x}_{d+1:2d}$$

then, this single filter $\mathbf{w}$ produces a single feature value (the output of some kind of pooling).

in practice, we use many filters of many different lengths (e.g., *n*-grams rather than words).

Convolutional Neural Networks

- “convolutional layer” = set of filters that are convolved with the input vector (whether ***x*** or hidden vector)
- could be followed by more convolutional layers, or by a type of pooling
- filters of varying n-gram lengths commonly used (1- to 5-grams)
- CNNs commonly used for character-level processing; filters look at character n-grams

- see demo