Natural Language Processing 1

Lecture 7: Compositional semantics and sentence representations

Katia Shutova and Sandro Pezzelle

ILLC
University of Amsterdam

18 November 2019

Outline.

Compositional semantics

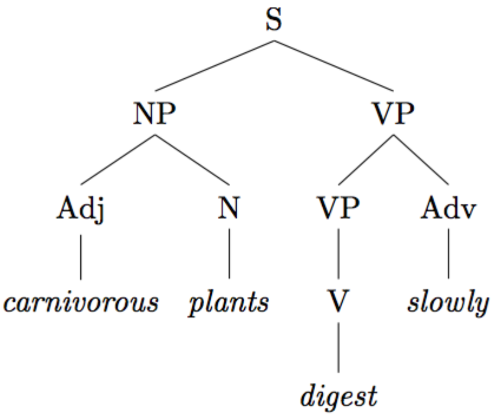
Compositional distributional semantics

Compositional semantics in neural networks

Compositional semantics

- Principle of Compositionality: meaning of each whole phrase derivable from meaning of its parts.
- Sentence structure conveys some meaning
- Deep grammars: model semantics alongside syntax, one semantic composition rule per syntax rule

Compositional semantics alongside syntax



Semantic composition is non-trivial

- Similar syntactic structures may have different meanings:

  it barks
  it rains; it snows – pleonastic pronouns

- Different syntactic structures may have the same meaning:

  Kim seems to sleep.
  It seems that Kim sleeps.

- Not all phrases are interpreted compositionally, e.g. idioms:

  red tape
  kick the bucket

  but they can be interpreted compositionally too, so we can not simply block them.

Semantic composition is non-trivial

- Elliptical constructions where additional meaning arises through composition, e.g. logical metonymy:

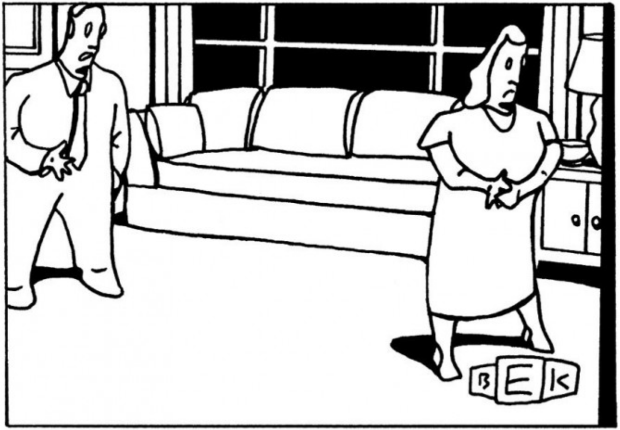
  fast programmer
  fast plane

- Meaning transfer and additional connotations that arise through composition, e.g. metaphor

  *I cant **buy** this story.*
  *This sum will **buy** you a ride on the train.*

- Recursion

Recursion



"Of course I care about how you imagined I thought you perceived I wanted you to feel."

Compositional semantic models

1. Compositional **distributional semantics**
   - model composition in a vector space
   - unsupervised
   - general-purpose representations
2. Compositional semantics in **neural networks**
   - supervised
   - (typically) task-specific representations

Outline.

Compositional semantics

Compositional distributional semantics

Compositional semantics in neural networks

Compositional distributional semantics

Can distributional semantics be extended to account for the meaning of phrases and sentences?

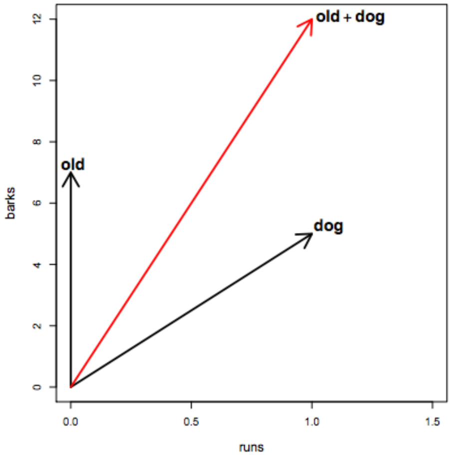
- Language can have an infinite number of sentences, given a limited vocabulary
- So we can not learn vectors for all phrases and sentences
- and need to do composition in a distributional space

1. Vector mixture models

Mitchell and Lapata, 2010. *Composition in Distributional Models of Semantics*

Models:

- Additive
- Multiplicative



Additive and multiplicative models

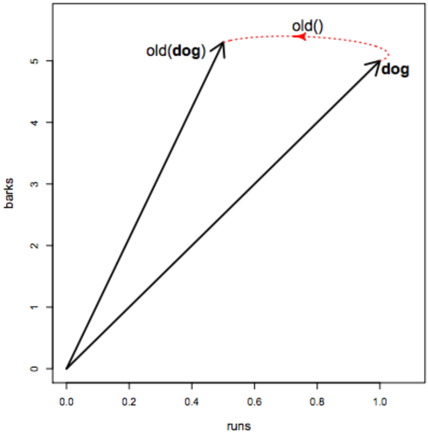
				additive		multiplicative	
	dog	**cat**	**old**	**old + dog**	**old + cat**	**old $\odot$ dog**	**old $\odot$ cat**
runs	1	4	0	1	4	0	0
barks	5	0	7	12	7	35	0

- correlate with human similarity judgments about adjective-noun, noun-noun, verb-noun and noun-verb pairs
- but... commutative, hence do not account for word order
  John hit the ball = *The ball hit John*!
- more suitable for modelling content words, would not port well to function words:
  e.g. *some dogs; lice and dogs; lice on dogs*

2. Lexical function models

Distinguish between:

- words whose meaning is directly determined by their distributional behaviour, e.g. nouns
- words that act as functions transforming the distributional profile of other words, e.g., verbs, adjectives and prepositions



Lexical function models

Baroni and Zamparelli, 2010. *Nouns are vectors, adjectives are matrices: Representing adjective-noun constructions in semantic space*

Adjectives as lexical functions

$$\textit{old dog} = \textit{old}(\textit{dog})$$

- Adjectives are parameter matrices ($\mathbf{A}_{old}$, $\mathbf{A}_{furry}$, etc.).
- Nouns are vectors (**house**, **dog**, etc.).
- Composition is simply **old dog** $=$ $\mathbf{A}_{old} \times$ **dog**.

$$\begin{array}{l|ll} \textbf{OLD} & \text{runs} & \text{barks} \\ \hline \text{runs} & 0.5 & 0 \\ \text{barks} & 0.3 & 1 \end{array} \times \begin{array}{l|l} & \textbf{dog} \\ \hline \text{runs} & 1 \\ \text{barks} & 5 \end{array} = \begin{array}{l|l} & \textbf{OLD(dog)} \\ \hline \text{runs} & (0.5 \times 1) + (0 \times 5) \\ & = 0.5 \\ \text{barks} & (0.3 \times 1) + (5 \times 1) \\ & = 5.3 \end{array}$$

Learning adjective matrices

For each adjective, learn a set of parameters that allow to predict the vectors of adjective-noun phrases

Training set:

house		**old house**
dog		**old dog**
car	$\rightarrow$	**old car**
cat		**old cat**
toy		**old toy**
...		**...**

Test set:

elephant	$\rightarrow$	**old elephant**
mercedes	$\rightarrow$	**old mercedes**

Learning adjective matrices

1. Obtain a distributional vector $\mathbf{n}_j$ for each noun n_j in the lexicon.
2. Collect adjective noun pairs (a_i, n_j) from the corpus.
3. Obtain a distributional vector $\mathbf{p}_{ij}$ of each pair (a_i, n_j) from the same corpus using a conventional DSM.
4. The set of tuples $\{(\mathbf{n}_j, \mathbf{p}_{ij})\}_j$ represents a dataset $\mathcal{D}(a_i)$ for the adjective a_i.
5. Learn matrix $\mathbf{A}_i$ from $\mathcal{D}(a_i)$ using linear regression.

Minimize the squared error loss:

$$L(\mathbf{A}_i) = \sum_{j \in \mathcal{D}(a_i)} \|\mathbf{p}_{ij} - \mathbf{A}_i \mathbf{n}_j\|^2$$

Verbs as higher-order tensors

Different patterns of subcategorization, i.e. how many (and what kind of) arguments the verb takes

- **Intransitive** verbs: only subject

  Kim slept

  modelled as a matrix (second-order tensor): $N \times M$

- **Transitive** verbs: subject and object

  Kim loves her dog

  modelled as a third-order tensor: $N \times M \times K$

Polysemy in lexical function models

Generally:

- use single representation for all senses
- assume that ambiguity can be handled as long as contextual information is available

Exceptions:

- Kartsaklis and Sadrzadeh (2013): homonymy poses problems and is better handled with prior disambiguation
- Gutierrez et al (2016): literal and metaphorical senses better handled by separate models
- However, this is still an open research question.

Modelling metaphor in lexical function models

Gutierrez et al (2016). *Literal and Metaphorical Senses in Compositional Distributional Semantic Models.*

- trained separate lexical functions for literal and metaphorical senses of adjectives
- mapping from literal to metaphorical sense as a linear transformation
- model can **identify metaphorical expressions**:

  e.g. brilliant person

- and **interpret** them

  brilliant person: clever person
  brilliant person: genius

Outline.

Compositional semantics

Compositional distributional semantics

Compositional semantics in neural networks

Compositional semantics and sentence representations w/ Neural Networks

Sandro Pezzelle
sandropezzelle.github.io
NLP1 2019. November 18, 2019

Credits: **Joost Bastings**

Overview

1) How do we learn a (task-specific) **representation** of a **sentence** with a **neural network**?

2) How do we make a **prediction** for a given **task** from that representation?

We will see the **task, dataset** and **models** of **Practical 2**!

Compositional Distributional Semantics vs Neural Networks

Compositional Distributional Semantics Models (cDSMs):

- *general-purpose* representations (e.g., sum of word embeddings)
- representations obtained in an *unsupervised* manner

Neural Networks (NNs):

- **task-specific** representations (i.e., optimized for one given task or set of tasks)
- representations learned in a **supervised** manner

Task

Task: Sentiment classification of movie reviews

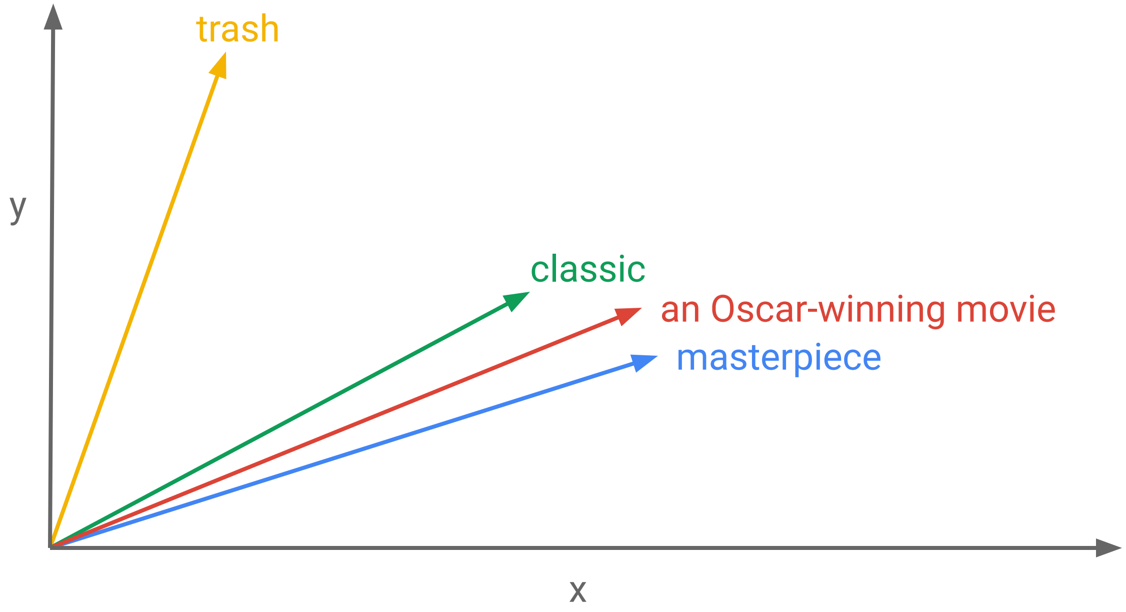
You'll probably love it. →

0. very negative
1. negative
2. neutral
3. **positive**
4. very positive

Task-specific: The learned representation has to be "specialized" on **sentiment**!

Words (and sentences) into vectors

When we talk about **representations** ...



Sentence representation: A (very) simplified picture

cDSMs (sum)	NNs
you	you
will	will
probably	probably
love	love
it	it

you will probably love it

you will probably love it

Dataset

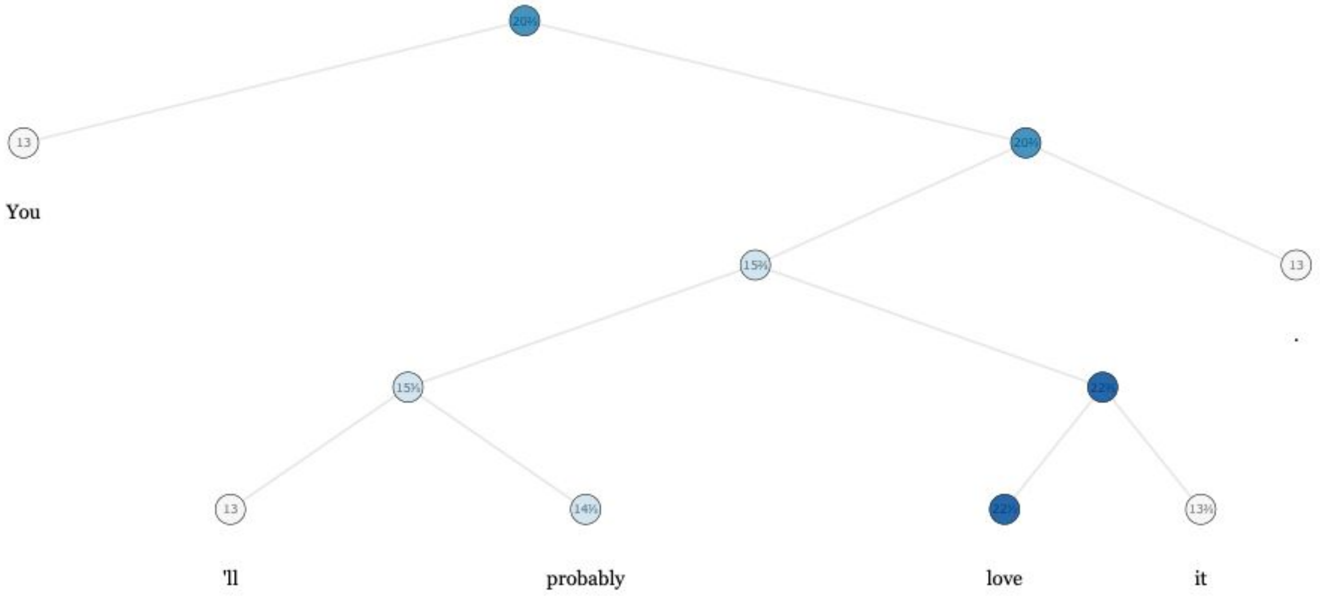
Dataset: Stanford Sentiment Treebank (SST)

11,855 data-points* including:

1. one-sentence review + “global” sentiment score
2. tree structure (syntax)
3. more detailed sentiment scores (node-level)

***Question:** Is this dataset big (for training Neural Nets)?

Binary parse tree: One example



Models

Models

Practical 2:
https://tinyurl.com/qrte8th

1. Bag of Words (BOW)
2. Continuous Bag of Words (CBOW)
3. Deep Continuous Bag of Words (Deep CBOW)
4. Deep CBOW + pre-trained word embeddings
5. LSTM
6. Tree LSTM

First approach: Sentence + Sentiment

1. **one-sentence review + “global” sentiment score**
2. tree structure (syntax)
3. node-level sentiment scores

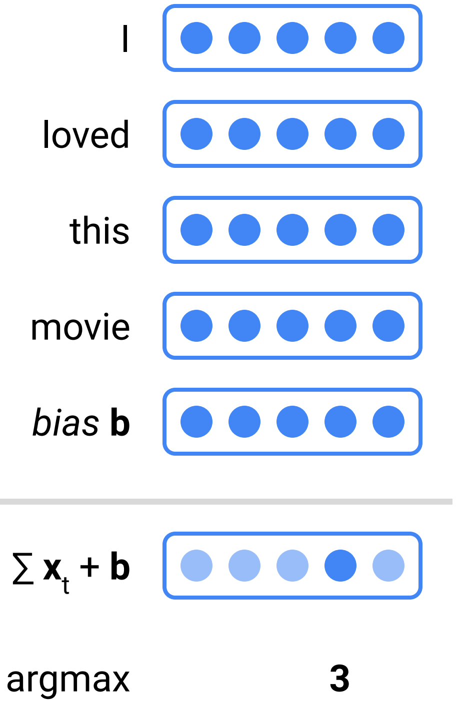
1. Bag of Words (BOW)

What is a Bag of Words?

Credits: CMU

Bag of Words

Sum word embeddings, add bias



Bag of Words

```
this   [0.0, 0.1, 0.1, 0.1, 0.0]
movie  [0.0, 0.1, 0.1, 0.2, 0.1]
is     [0.0, 0.1, 0.0, 0.0, 0.0]
stupid [0.9, 0.5, 0.1, 0.0, 0.0]

bias   [0.0, 0.0, 0.0, 0.0, 0.0]
--------------------------------
sum    [0.9, 0.8, 0.3, 0.3, 0.1]

    argmax: 0 (very negative)
```

Turning words into numbers

We want to **feed words** to a neural network
How to turn **words** into **numbers**?

Bad idea: number sequence

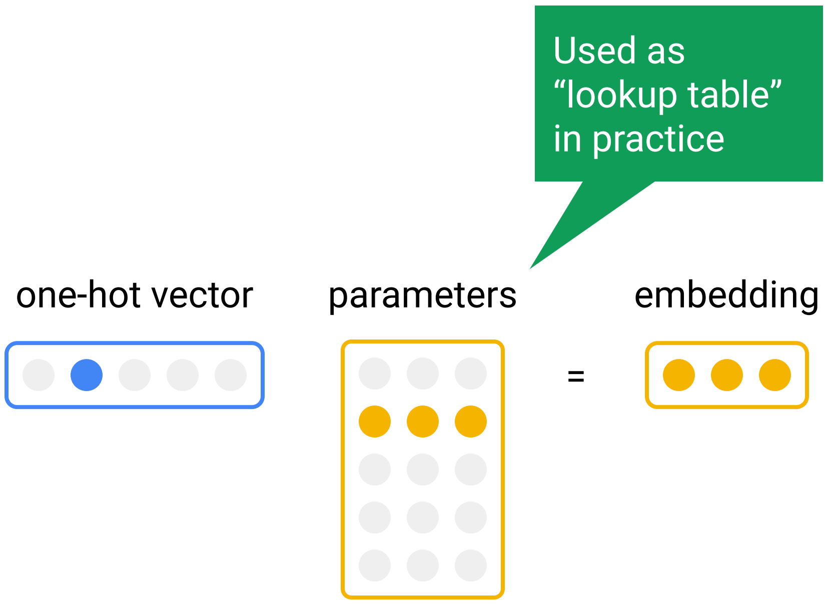
```
cat   1
tree  2
chair 3
dog   4
mat   5
```

Good idea: one-hot vectors

```
cat   [0, 0, 0, 0, 1]
tree  [0, 0, 0, 1, 0]
chair [0, 0, 1, 0, 0]
dog   [0, 1, 0, 0, 0]
mat   [1, 0, 0, 0, 0]
```

cat is closer to **tree**
than to **dog**?!

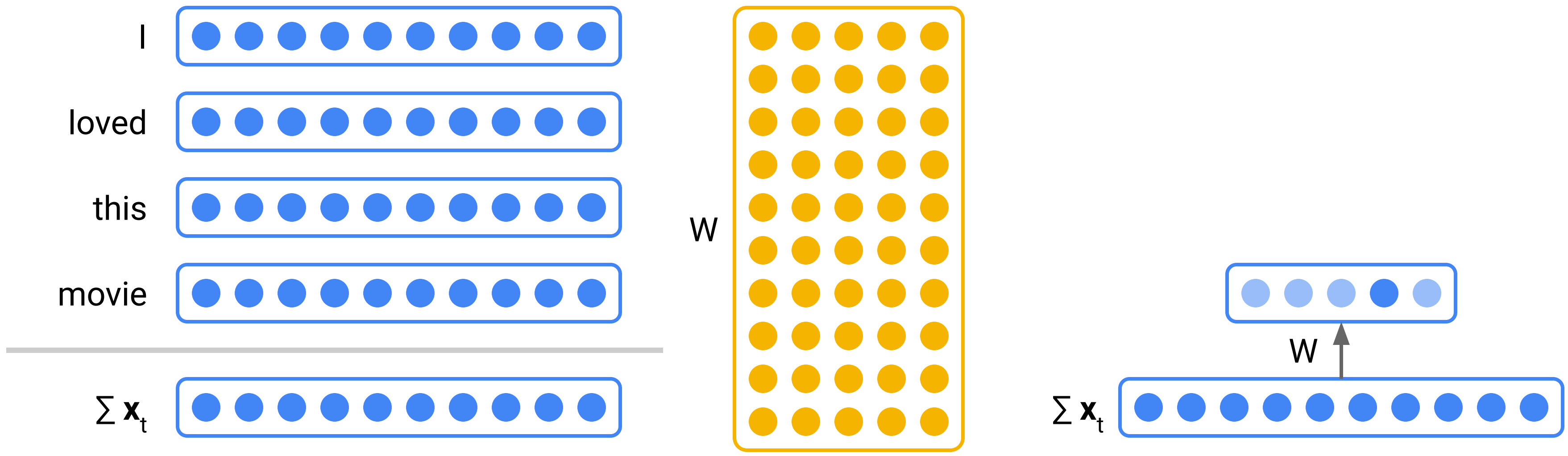
One-hot vectors select word embeddings



2. Continuous Bag of Words (CBOW)

Continuous Bag of Words (CBOW)

Sum word embeddings, project to 5D using W, add bias: W ($\sum \mathbf{x}_t$) + **b**



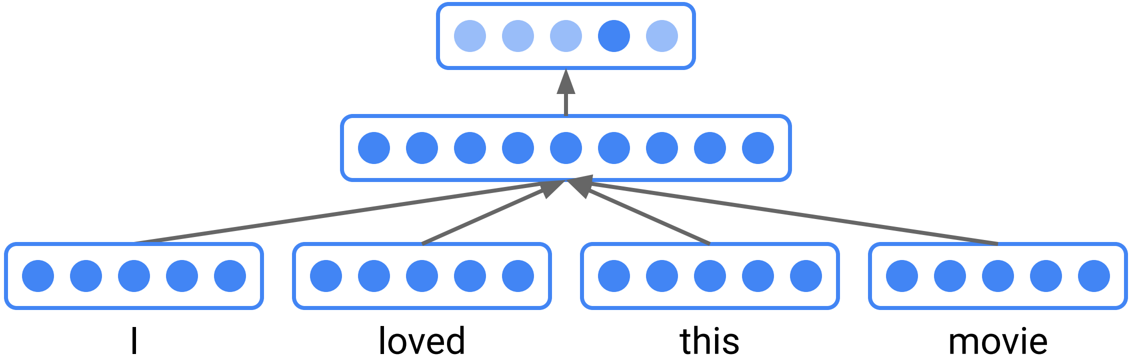
Note that a bias term (of size 5) is added to the final output vector (not shown). Also, this is not the same as word2vec CBOW!

Recall: Matrix Multiplication

Rows multiply with **columns**

$$\underset{2\times 3}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}} \times \underset{3\times 2}{\begin{bmatrix} 1 & 2 \\ 1 & 2 \\ 1 & 2 \end{bmatrix}} = \underset{2\times 2}{\begin{bmatrix} 1\times 1 + 2\times 1 + 3\times 1 & 1\times 2 + 2\times 2 + 3\times 2 \\ 4\times 1 + 5\times 1 + 6\times 1 & 4\times 2 + 5\times 2 + 6\times 2 \end{bmatrix}}$$

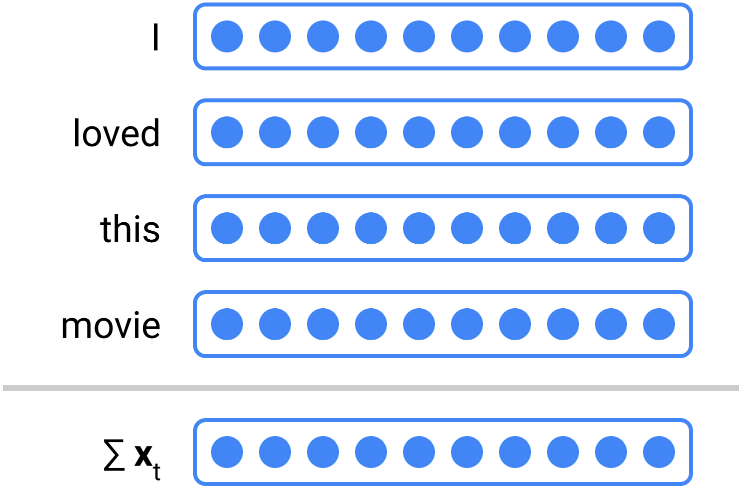
What about this?

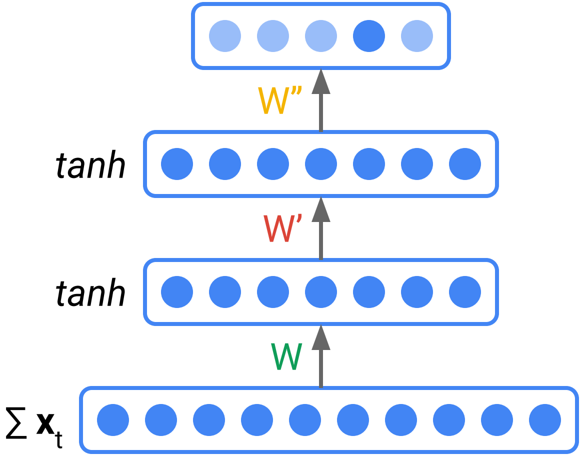


3. Deep CBOW

Deep CBOW

$$W'' \tanh\Big(W' \tanh(W (\sum \mathbf{x}_t) + \mathbf{b}) + \mathbf{b}' \Big) + \mathbf{b}''$$



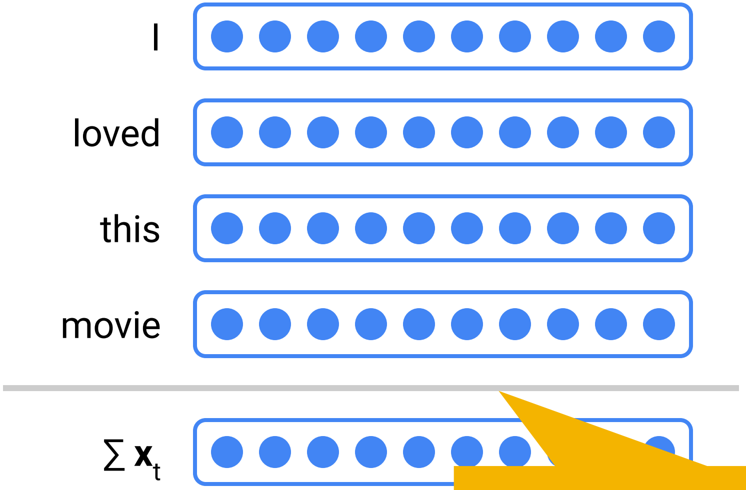


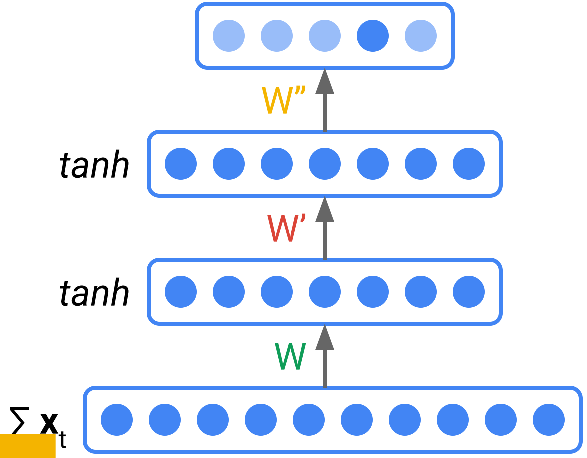
Note that a bias term is added whenever we multiply with a W (not shown)

4. Deep CBOW + Pretrained embeddings

Deep CBOW with pretrained embeddings

$$W'' \tanh\left(W' \tanh(W (\sum \mathbf{x}_t) + \mathbf{b}) + \mathbf{b}' \right) + \mathbf{b}''$$





Instead of learning them from scratch, feed word2vec or Glove embeddings!

Note that a bias term is added whenever we multiply with a W (not shown)

Deep CBOW with pretrained embeddings

Question: Why would that help?

Recap: Training a neural network

We train our network with Stochastic Gradient Descent (SGD):

1. Sample a training example
2. Forward pass
    a. Compute network activations, output vector
3. Compute loss
    a. Compare output vector with true label using a **loss function (Cross Entropy)**
4. Backward pass (backpropagation)
    a. Compute gradient of loss w.r.t. (learnable) parameters (= weights + bias)
5. Take a small step in the opposite direction of the gradient

Cross Entropy Loss

Given:

$\hat{\mathbf{y}}$ = [`0.0589, 0.0720, 0.0720,` **`0.7177`**`, 0.0795`] *output vector (after* ***softmax****) from forward pass*

$\mathbf{y}$ = [`  0,    0,    0,    1,    0`] *target / label (y_3 = 1)*

When our output is **categorical** (i.e., a number of classes), we can use a **Cross Entropy** loss:

$\mathrm{CE}(\mathbf{y}, \hat{\mathbf{y}}) = - \sum y_i \log \hat{y}_i$

$\mathrm{SparseCE}(y = 3, \hat{\mathbf{y}}) = - \log \hat{y}_y$

`torch.nn.CrossEntropyLoss` works like this and does the **softmax** on **o** for you!

Softmax

We don't need a softmax for **prediction**, there we simply take the **argmax**

$$\mathbf{o} = [-0.1,\ 0.1,\ 0.1,\ \mathbf{2.4},\ 0.2]$$

But we do need a **softmax** combined to CE to compute model loss (argmax is NOT differentiable)

$$\textit{softmax}(o_i) = \exp(o_i) \,/\, \textstyle\sum_j \exp(o_j)$$

This makes **o** sum to 1.0:

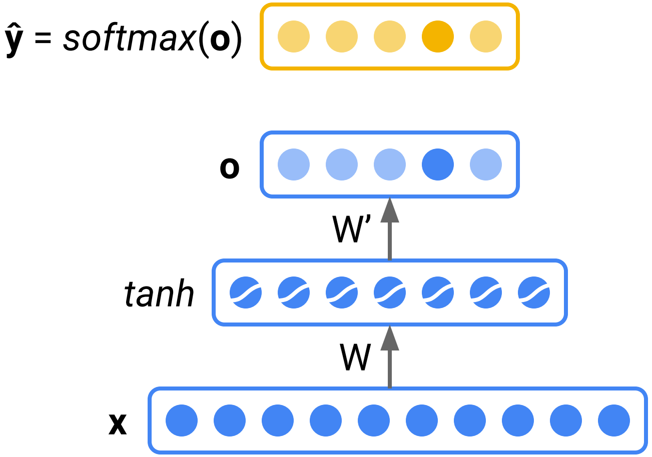
$$\text{softmax}(\mathbf{o}) = [0.0589,\ 0.0720,\ 0.0720,\ \mathbf{0.7177},\ 0.0795]$$

Backpropagation example

the **chain rule** is your friend!

$L = f(g(x))$

$\delta L/\delta x = \delta f(g(x))/\delta g(x) \cdot \delta g(x)/\delta x$



$\hat{\mathbf{y}} = [0.0589,\ 0.0720,\ 0.0720,\ \mathbf{0.7177},\ 0.0795]$

$\mathbf{y} = [\ 0,\ 0,\ 0,\ 1,\ 0]$

loss $L = CE(\hat{\mathbf{y}}, \mathbf{y}) = -\log(\hat{y}_3) = -\log(0.7177)$

$= \mathbf{0.144}$

compute gradients, e.g. for W':

$\delta L/\delta W' = \delta L/\delta o\ \ \delta o/\delta W'$

$\delta L/\delta o = \delta L/\delta \hat{y}\ \ \delta \hat{y}/\delta o$

$= -1/\hat{y}_3\ \ \delta softmax(o)/\delta o$

update weights:

$W' = W' - eta * \delta L/\delta W'$

Recurrent Neural Networks

Introduction: Recurrent Neural Network (RNN)

- RNNs widely used for handling **sequences**!
- RNNs ~ **multiple copies of same network**, each passing a message to a successor
- Take an input vector *x* and output an output vector *h*
- Crucially, *h* **influenced by entire history** of inputs fed in in the past
- Internal state *h* gets updated at every time step → in the simplest case, this state consists of a **single hidden vector *h***

Introduction: Recurrent Neural Network (RNN)

Example:

the	cat	sat	on	the	mat
$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$	$\mathbf{x}_4$	$\mathbf{x}_5$	$\mathbf{x}_6$

Let's compute the RNN state after reading in this sentence.

Remember:

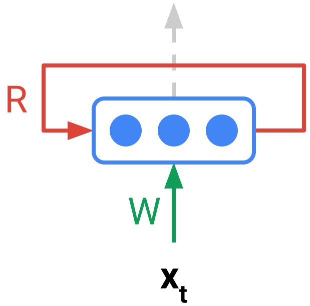
$\mathbf{h}_t = f(\, \mathbf{x}_t, \mathbf{h}_{t-1}\,)$

$$\mathbf{h}_1 = f(\mathbf{x}_1, \mathbf{h}_0)$$
$$\mathbf{h}_2 = f(\mathbf{x}_2, f(\mathbf{x}_1, \mathbf{h}_0))$$
$$\mathbf{h}_3 = f(\mathbf{x}_3, f(\mathbf{x}_2, f(\mathbf{x}_1, \mathbf{h}_0)))$$
…
$$\mathbf{h}_6 = f(\mathbf{x}_6, f(\mathbf{x}_5, f(\mathbf{x}_4, \ldots)))$$

Elman (1990)

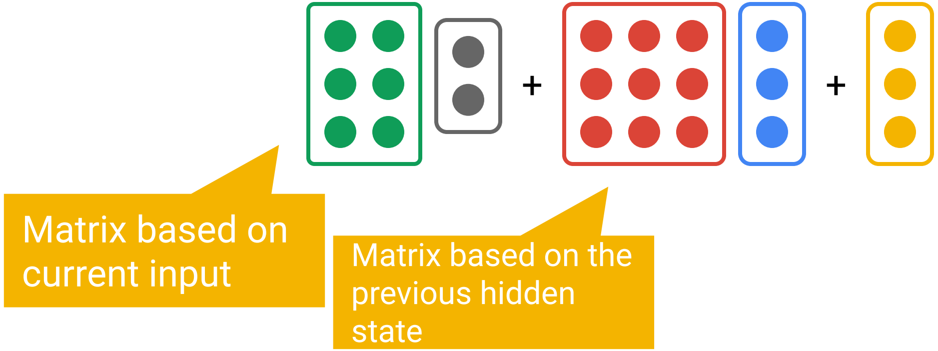
Introduction: Recurrent Neural Network (RNN)

RNNs model **sequential data** - one input $\mathbf{x}_t$ per time step *t*



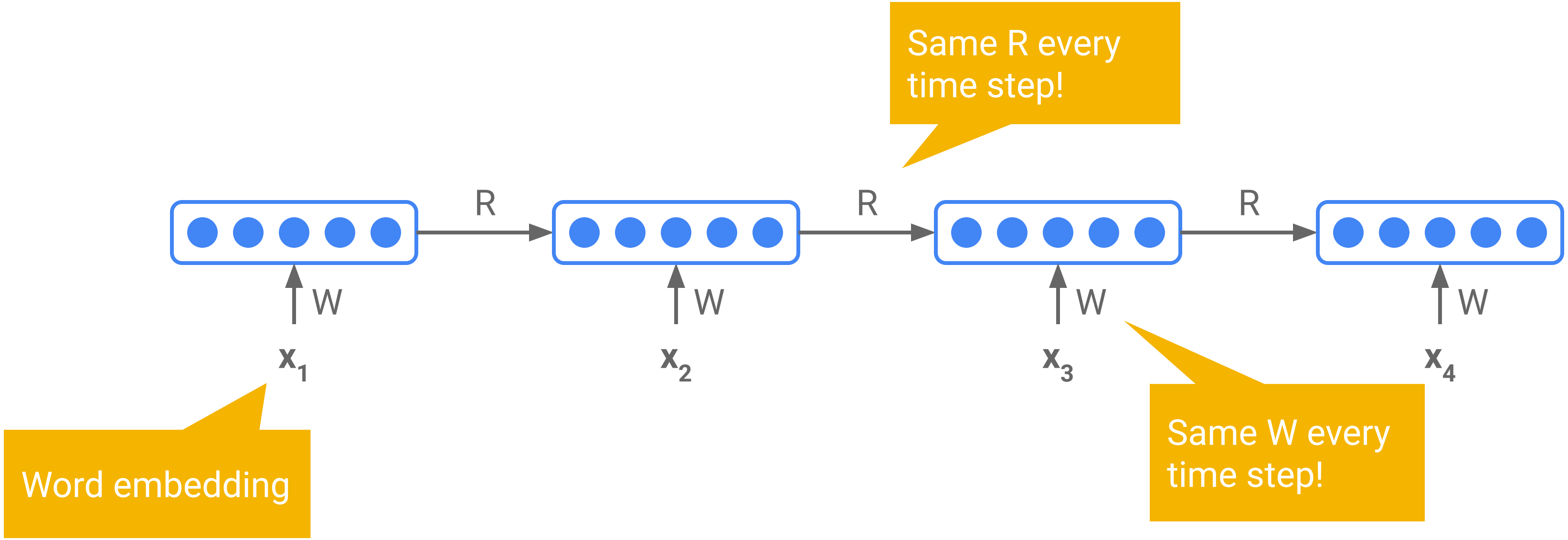
$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1})$$

$$= \sigma(W\mathbf{x}_t + R\mathbf{h}_{t-1} + \mathbf{b})$$

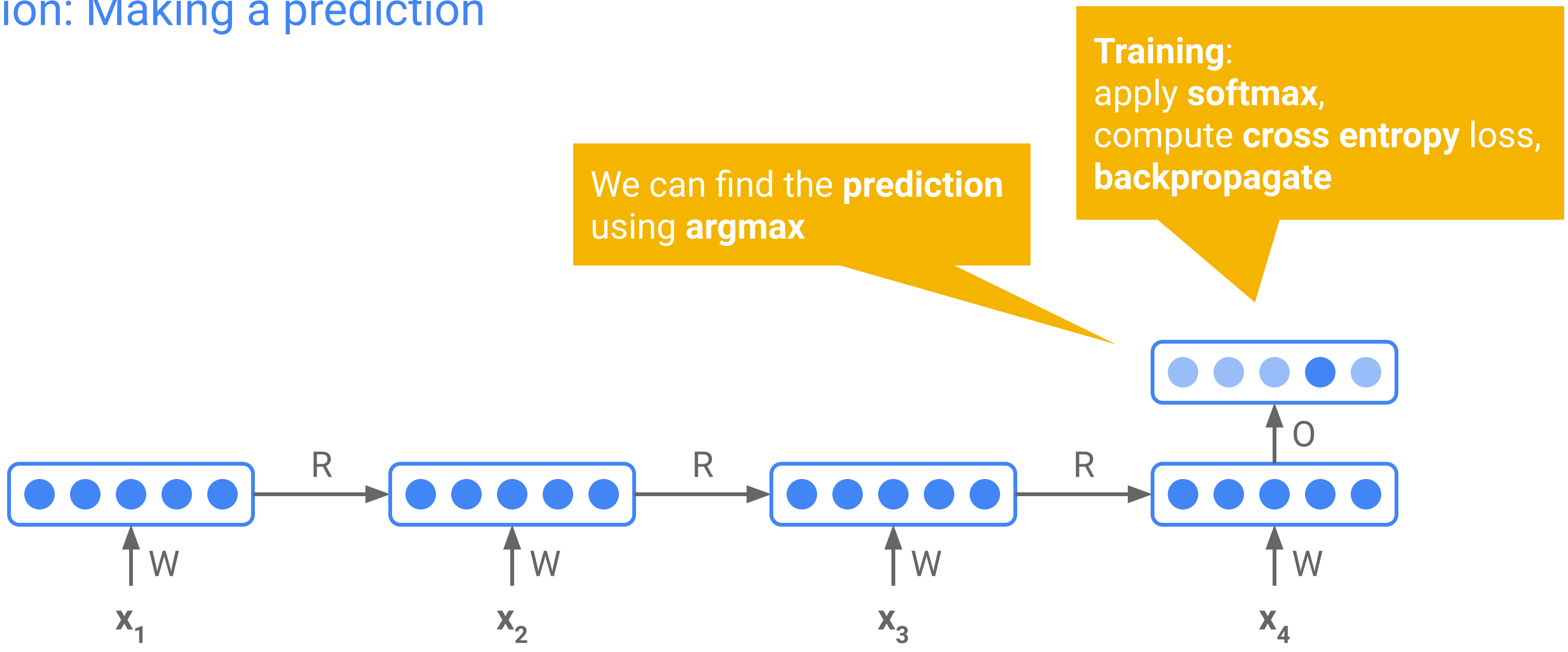


Elman (1990). Finding structure in time.

Introduction: Unfolding the RNN



Introduction: Making a prediction

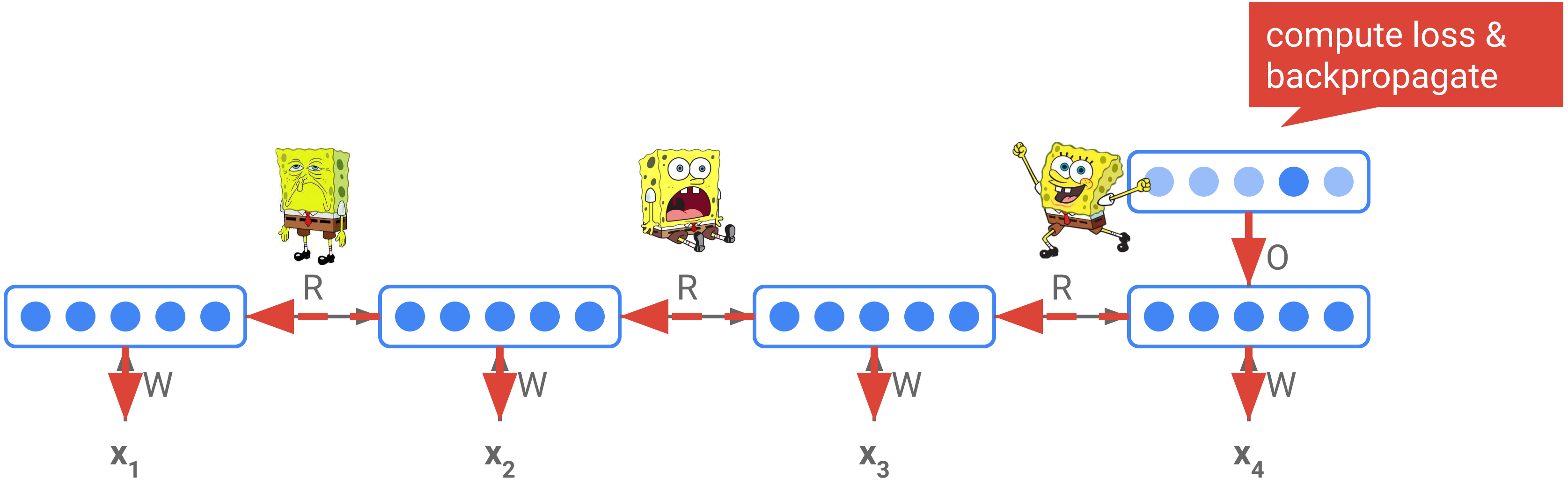


Introduction: The vanishing gradient problem

Simple RNNs are hard to train because of the **vanishing gradient** problem.

During backpropagation, **gradients** can quickly become **small**,

as they **repeatedly** go through multiplications (R) & non-linear functions (e.g. sigmoid or tanh)

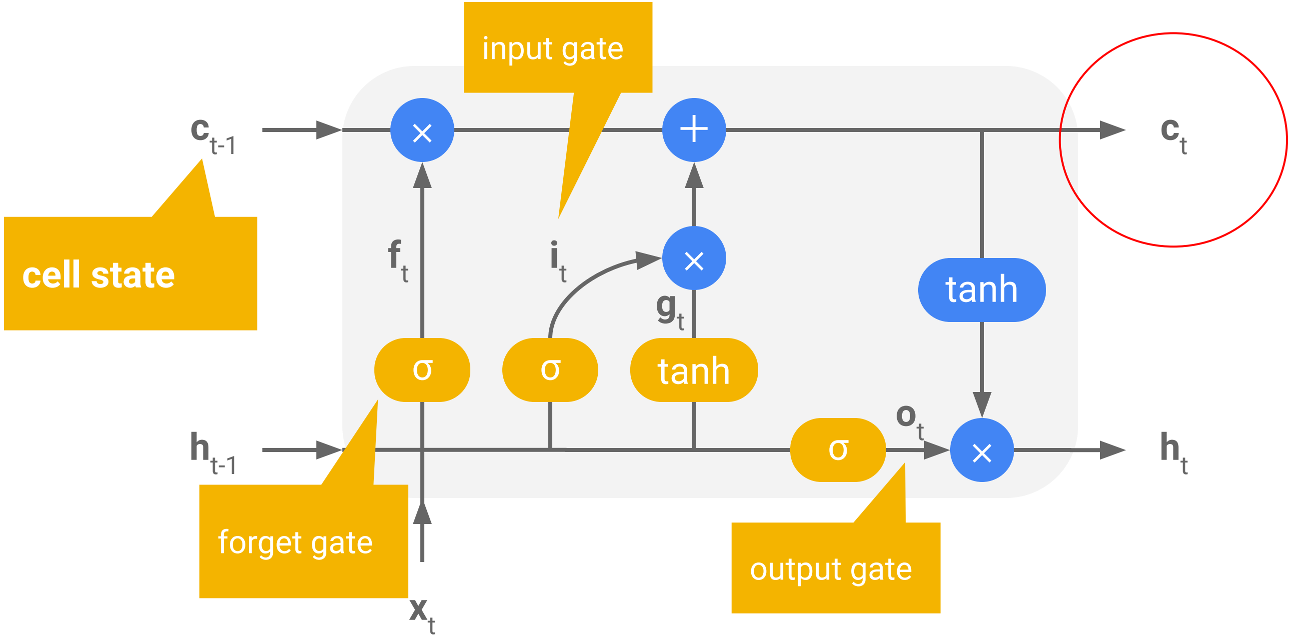


For more details see: Kyunghyun Cho. Natural Language Understanding with Distributed Representation. Section 4.3.

5. Long Short-Term Memory network (LSTM)

Long Short-Term Memory (LSTM)

LSTMs are a special kind of RNN that can deal with **long-term dependencies** in the data

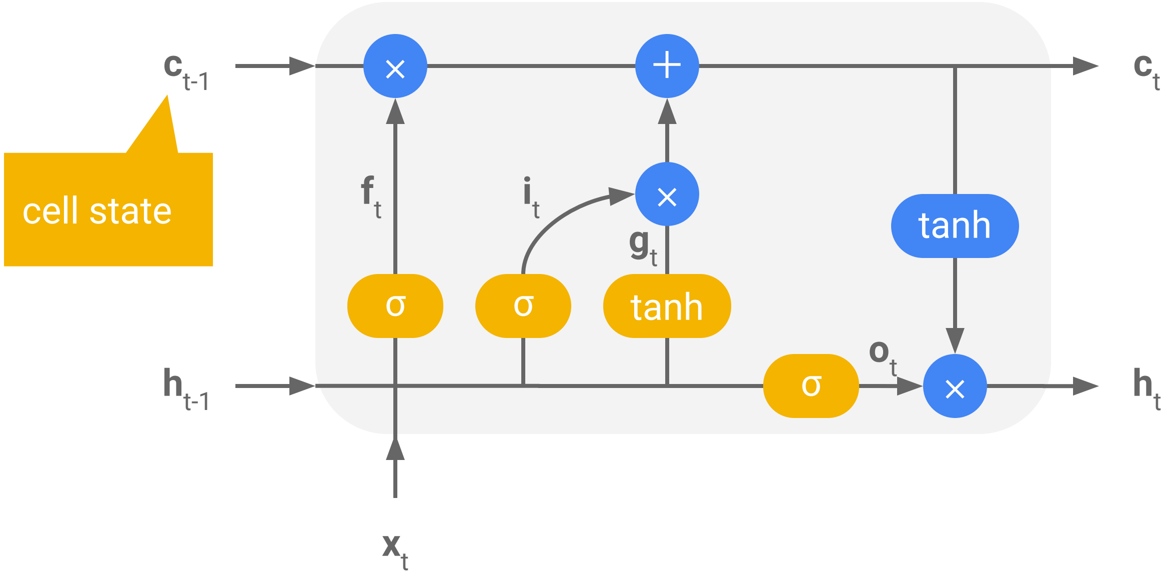


Adapted from http://colah.github.io/posts/2015-08-Understanding-LSTMs . Yellow blocks: $\phi(W[\mathbf{h}_{t-1};\mathbf{x}_t] + \mathbf{b})$, blue blocks: element-wise operation

Long Short-Term Memory (LSTM): Core idea

CELL STATE: “conveyor belt”. It runs straight down the entire chain, with only some minor linear interactions. Information can just flow along it unchanged. LSTM can remove or add information to the cell state, carefully regulated by structures called gates

Long Short-Term Memory (LSTM)

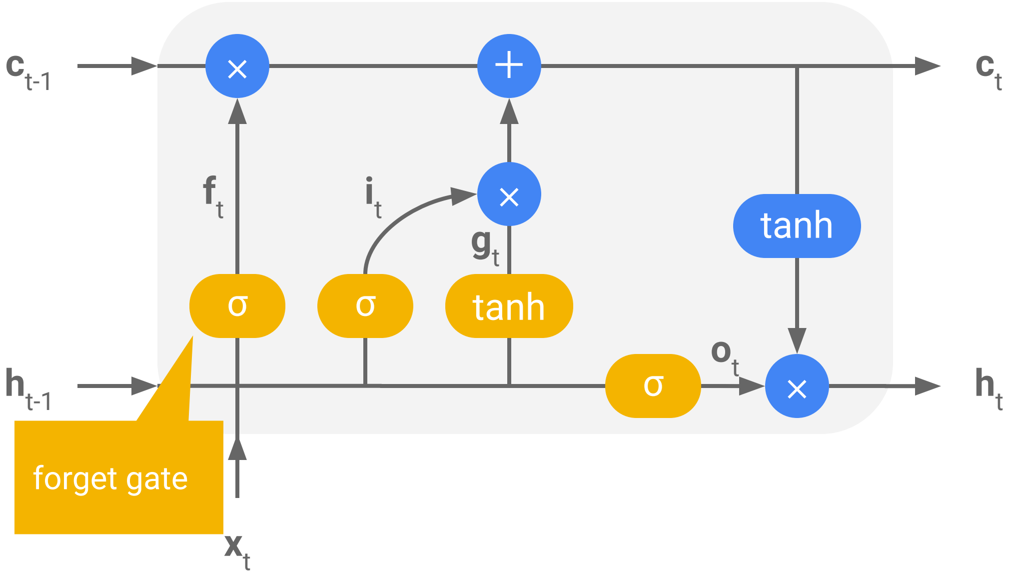


Adapted from http://colah.github.io/posts/2015-08-Understanding-LSTMs . Yellow blocks: $\phi(W[\mathbf{h}_{t-1};\mathbf{x}_t] + \mathbf{b})$, blue blocks: element-wise operation

Long Short-Term Memory (LSTM): Step 1

Decide what information to throw away from the cell state: **FORGET GATE** looks at h_{t-1} and x_t and outputs a number between 0 and 1 (sigmoid) for each value in the cell state C_{t-1}. 1 represents “completely keep this”; a 0 represents “completely get rid of this”

Long Short-Term Memory (LSTM)

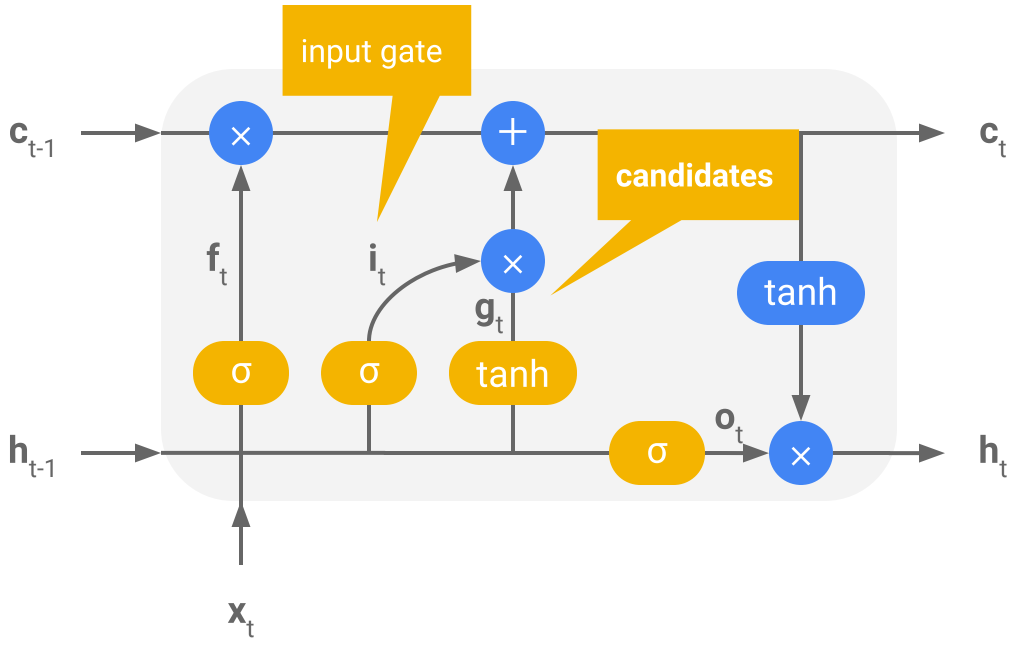


Adapted from http://colah.github.io/posts/2015-08-Understanding-LSTMs . Yellow blocks: $\phi(W[\mathbf{h}_{t-1};\mathbf{x}_t] + \mathbf{b})$, blue blocks: element-wise operation

Long Short-Term Memory (LSTM): Step 2

Decide what new information to store in the cell state. Two steps: (1) a sigmoid layer (**INPUT GATE**) decides which values we update (looks at x_t and h_{t-1}). (2) A *tanh* layer [-1,1] creates a vector of new **candidate values**, g_t, that could be added to the cell state

Long Short-Term Memory (LSTM)

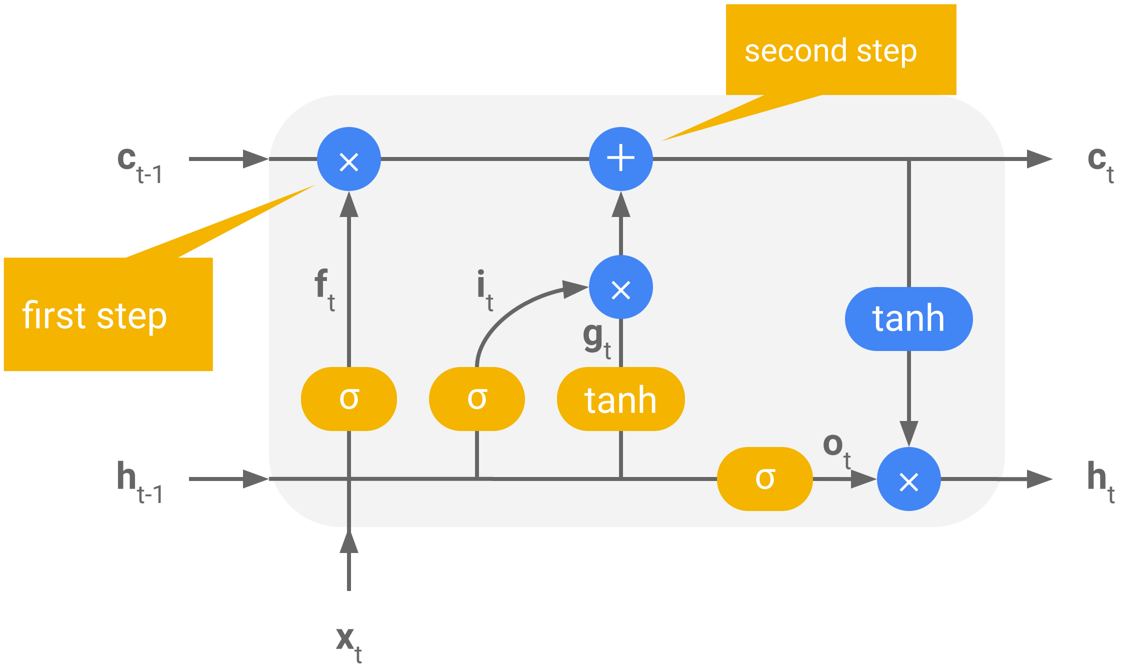


Adapted from http://colah.github.io/posts/2015-08-Understanding-LSTMs . Yellow blocks: $\phi(W[\mathbf{h}_{t-1};\mathbf{x}_t] + \mathbf{b})$, blue blocks: element-wise operation

Long Short-Term Memory (LSTM): Step 3

Update the old cell state, C_{t-1}, into the new cell state C_t: The old state is multiplied by **FORGET LAYER** f_t, forgetting the things we decided to forget earlier. Then we add **INPUT LAYER * CANDIDATE VALUES** ($i_t * g_t$). This are the new candidate values scaled by how much we decided to update each state value

Long Short-Term Memory (LSTM)

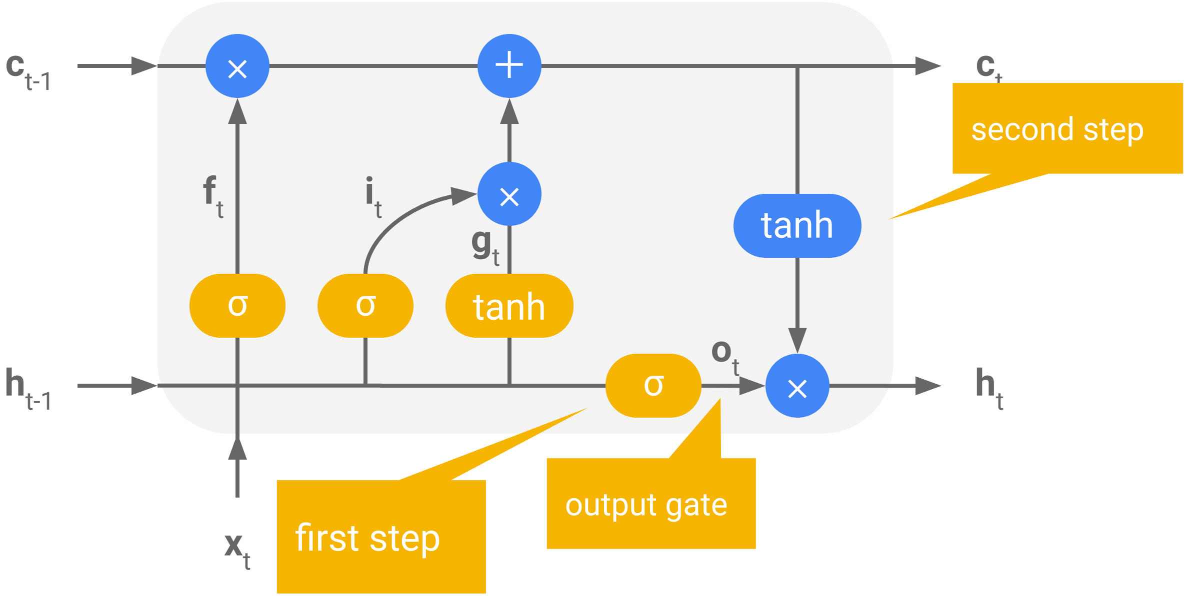


Adapted from http://colah.github.io/posts/2015-08-Understanding-LSTMs . Yellow blocks: $\phi(W[\mathbf{h}_{t-1};\mathbf{x}_t] + \mathbf{b})$, blue blocks: element-wise operation

Long Short-Term Memory (LSTM): Output

Decide what to output: First, a sigmoid layer (**OUTPUT GATE**) decides what parts of the cell state we're going to output. Then, the cell state is put through *tanh* [−1,1] and multiplied by the output of the output gate, so that we only output the parts we decided to

Long Short-Term Memory (LSTM)



Adapted from http://colah.github.io/posts/2015-08-Understanding-LSTMs . Yellow blocks: $\phi(W[\mathbf{h}_{t-1};\mathbf{x}_t] + \mathbf{b})$, blue blocks: element-wise operation

Long Short-Term Memory (LSTM)

hidden state | cell state | previous hidden state and cell state

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

input gate	$\mathbf{i}_t$	=	$\sigma(W_i\,\mathbf{x}_t + R_i\,\mathbf{h}_{t-1} + \mathbf{b}_i)$
forget gate	$\mathbf{f}_t$	=	$\sigma(W_f\,\mathbf{x}_t + R_f\,\mathbf{h}_{t-1} + \mathbf{b}_f)$
candidate	$\mathbf{g}_t$	=	$\tanh(W_g\,\mathbf{x}_t + R_g\,\mathbf{h}_{t-1} + \mathbf{b}_g)$
output gate	$\mathbf{o}_t$	=	$\sigma(W_o\,\mathbf{x}_t + R_o\,\mathbf{h}_{t-1} + \mathbf{b}_o)$

cell state	$\mathbf{c}_t$	=	$\mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t$
hidden state	$\mathbf{h}_t$	=	$\mathbf{o}_t \odot \tanh(\mathbf{c}_t)$

LSTMs: Applications & Success in NLP

- Language modeling (Mikolov et al., 2010; Sundermeyer et al., 2012)
- Parsing (Vinyals et al., 2015; Kiperwasser and Goldberg, 2016; Dyer et al., 2016)
- Machine translation (Bahdanau et al., 2015)
- Image captioning (Bernardi et al., 2016)
- Visual question answering (Antol et al., 2015)
- … and many other tasks!

Trees

Second approach: Sentence + Sentiment + Syntax

1. one-sentence review + “global” sentiment score
2. **tree structure (syntax)**
3. node-level sentiment scores

Exploiting tree structure

Instead of treating our input as a **sequence**, we can take an alternative approach: assume a **tree structure** and use the principle of **compositionality**.

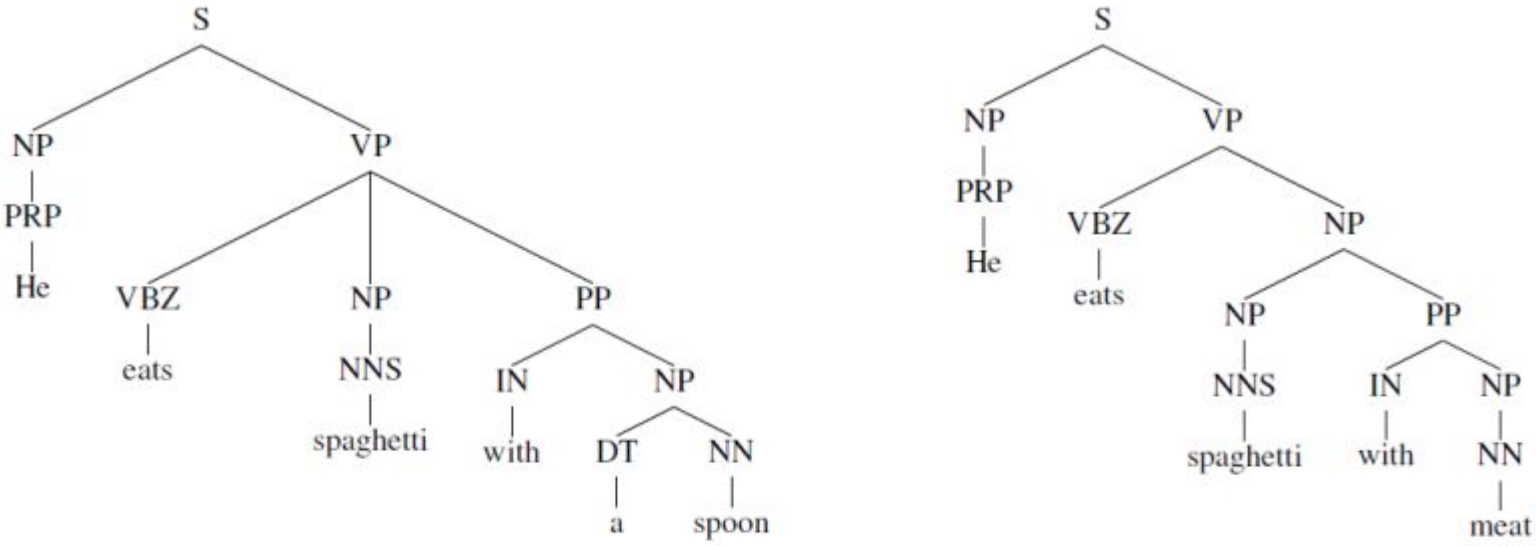
The meaning (vector) of a sentence is determined by:

1. the meanings of its **words** and
2. the **rules** that combine them

Adapted from Stanford cs224n.

Why would it be useful?

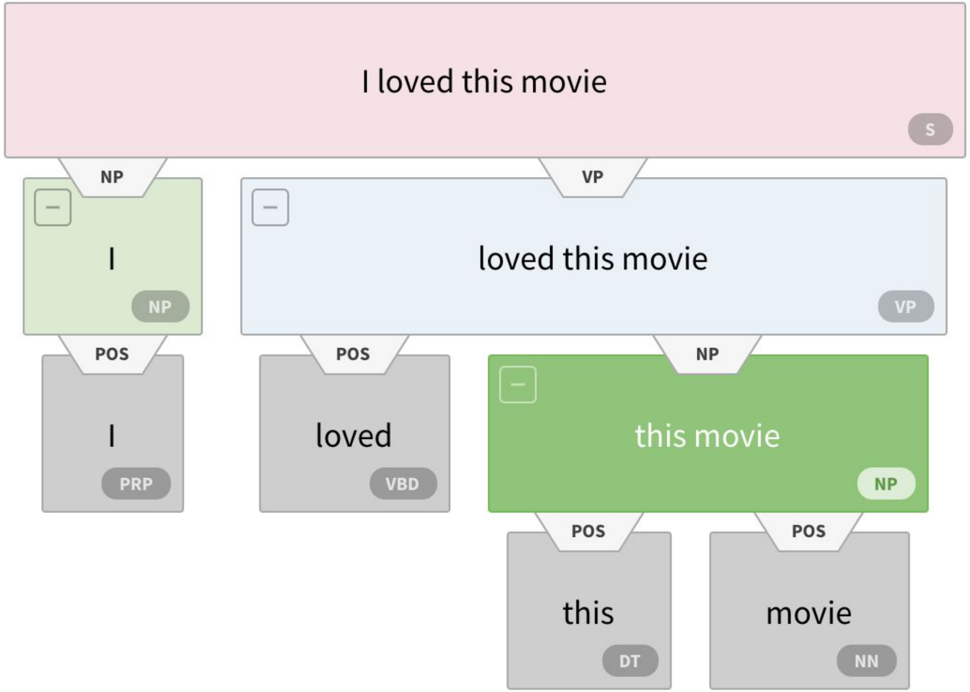
Helpful in **disambiguation**: similar “surface” / different structure



Credits: https://cs224d.stanford.edu/lectures/CS224d-Lecture10.pdf

Constituency Parse

Can we obtain a **sentence vector** using the tree structure given by a parse?

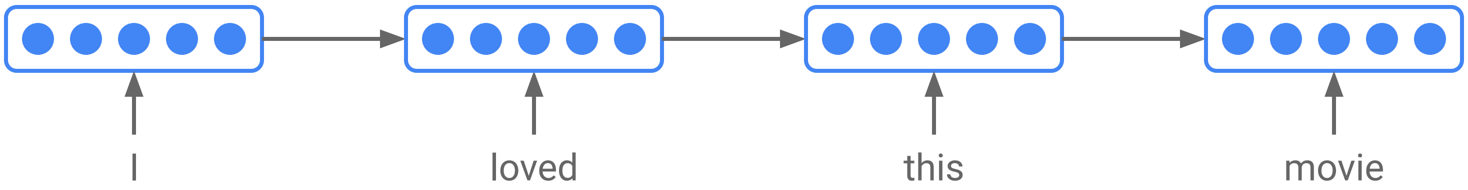


http://demo.allennlp.org/constituency-parsing

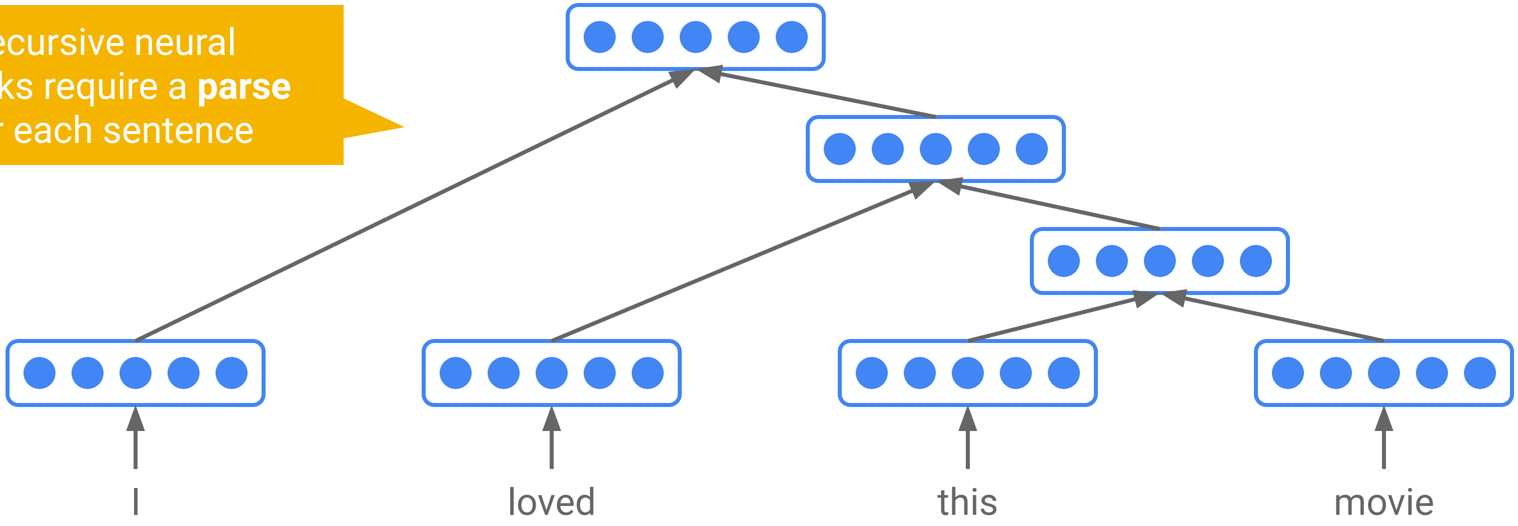
6. Tree LSTM

Recurrent vs Tree Recursive NN

RNNs cannot capture phrases **without prefix context** and often capture too much of **last words** in final vector

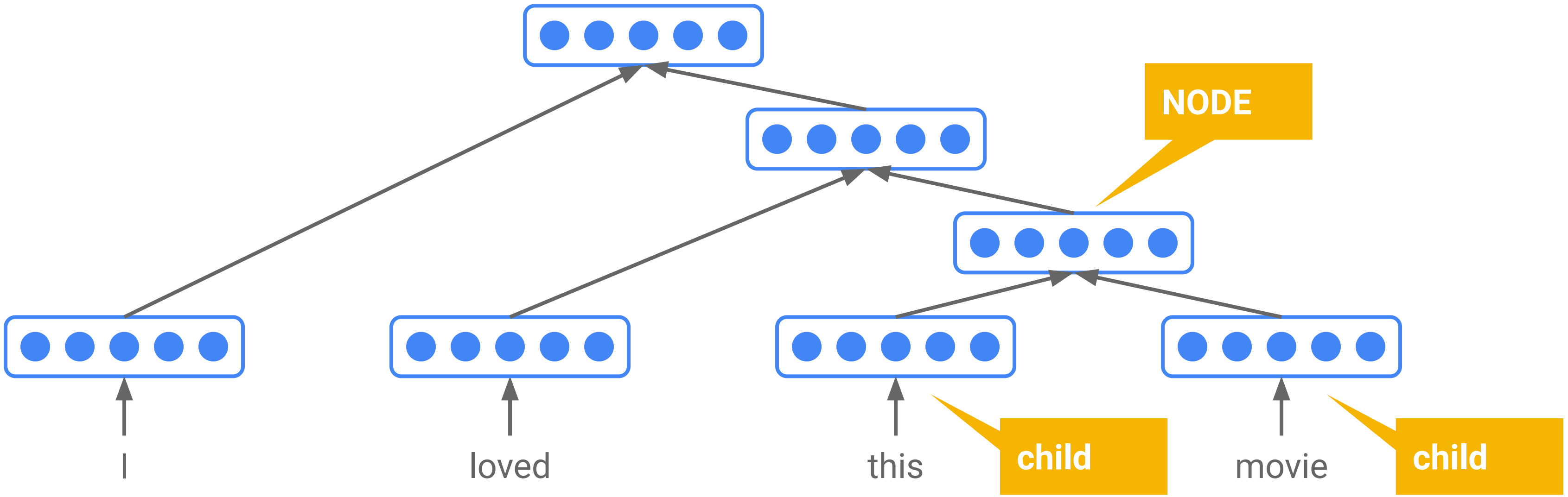


Tree Recursive neural networks require a **parse tree** for each sentence



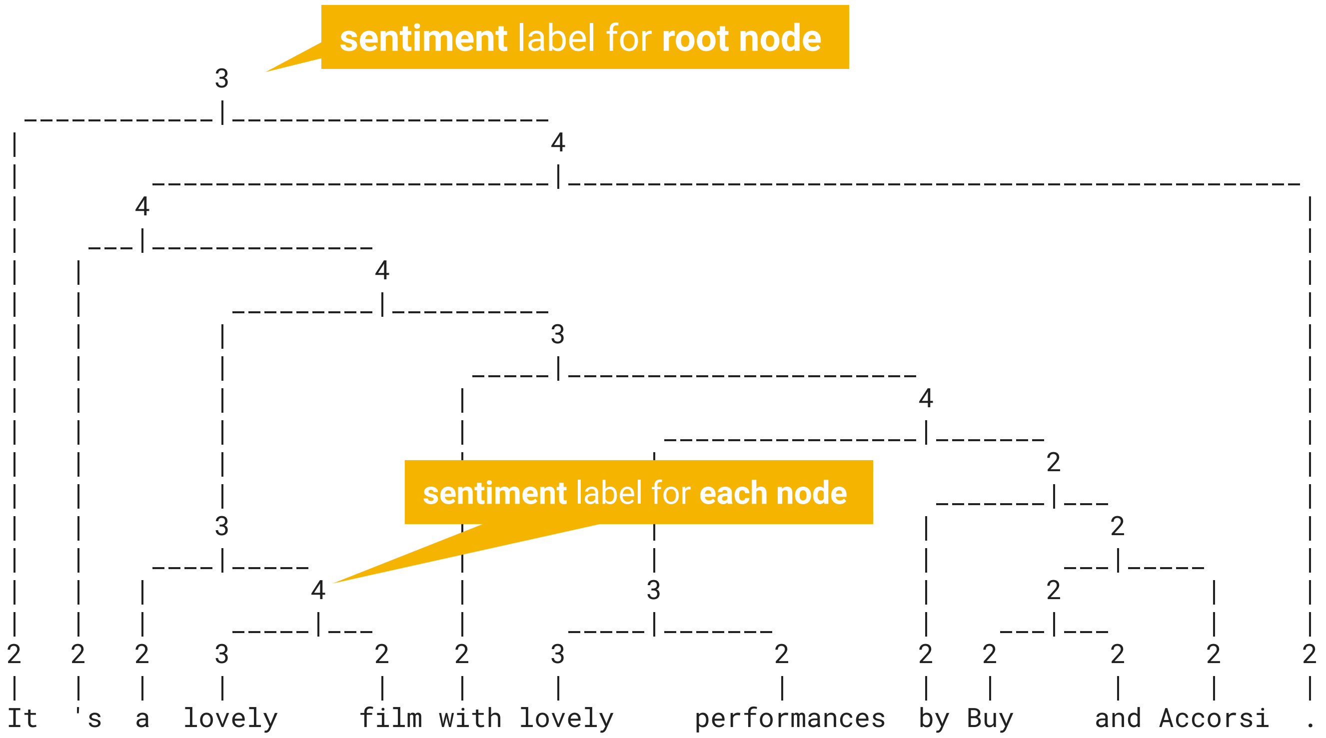
Adapted from Stanford cs224n.

Tree Recursive NN



Adapted from Stanford cs224n.

Practical II data set: Stanford Sentiment Treebank (SST)



Tree LSTMs: Generalize LSTM to tree structure

Use the idea of LSTM (gates, memory cell) but allow for multiple inputs (**node children**)

Proposed by 3 groups in the same summer:

- Kai Sheng Tai, Richard Socher, and Christopher D. Manning. ***Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks***. ACL 2015.
    - Child-Sum Tree LSTM
    - N-ary Tree LSTM
- Phong Le and Willem Zuidema. *Compositional distributional semantics with long short term memory.* *SEM 2015.
- Xiaodan Zhu, Parinaz Sobihani, and Hongyu Guo. *Long short-term memory over recursive structures*. ICML 2015.

Tree LSTMs

1. Child-Sum Tree LSTM

   sums over all children of a node; can be used for any N of children

2. N-ary Tree LSTM

   different parameters for each child; better granularity (interactions between children)

   but maximum N of children per node has to be fixed

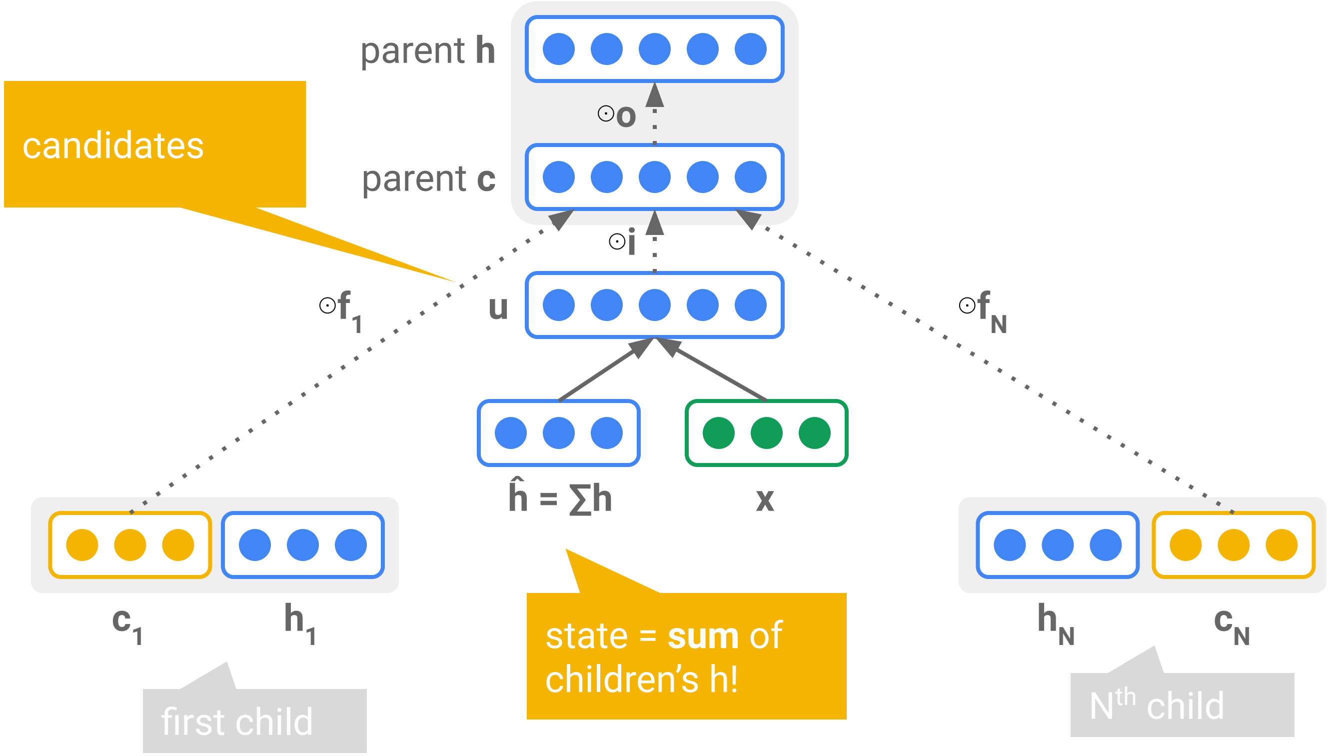
Credits: Daniel Perez https://www.slideshare.net/tuvistavie/tree-lstm

Child-Sum Tree LSTM

Children **outputs** and **memory cells** are **summed**

1. NO children order
2. works with variable number of children (sum!)
3. shares gates weights between children

Child-Sum Tree LSTM



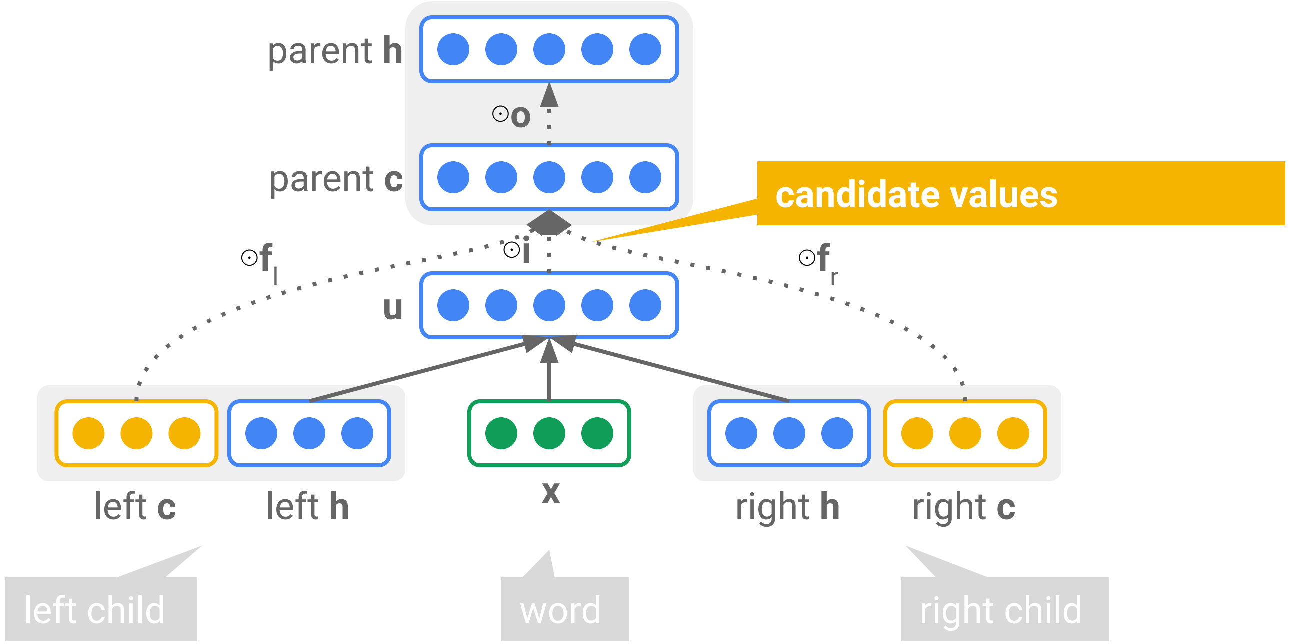
N-*ary* Tree LSTM

Implemented
in **Practical 2**

Separate parameter matrices for each child *k*

1. each node must have at most N (e.g., **binary**) ordered children
2. fine-grained control on how information propagates
3. forget gate can be parametrized (N matrices, one per *k*) so that siblings affect each other

N-ary Tree LSTM



N-ary Tree LSTM

useful for encoding **constituency** trees

$$i_j = \sigma\left(W^{(i)}x_j + \sum_{\ell=1}^{N} U_\ell^{(i)} h_{j\ell} + b^{(i)}\right),$$

$$f_{jk} = \sigma\left(W^{(f)}x_j + \sum_{\ell=1}^{N} U_{k\ell}^{(f)} h_{j\ell} + b^{(f)}\right),$$

$$o_j = \sigma\left(W^{(o)}x_j + \sum_{\ell=1}^{N} U_\ell^{(o)} h_{j\ell} + b^{(o)}\right),$$

$$u_j = \tanh\left(W^{(u)}x_j + \sum_{\ell=1}^{N} U_\ell^{(u)} h_{j\ell} + b^{(u)}\right),$$

$$c_j = i_j \odot u_j + \sum_{\ell=1}^{N} f_{j\ell} \odot c_{j\ell},$$

$$h_j = o_j \odot \tanh(c_j),$$

LSTMs vs Tree-LSTMs

Question: Can standard LSTMs be considered as (a special case of) Tree-LSTMs?

Transition Sequence Representation

Building a tree with a transition sequence

We can describe a **binary tree** using a *shift-reduce* **transition sequence**

practical II explains how to obtain this sequence

```
(I ( loved ( this movie ) ) )
 S   S       S    S      R  R  R
```

We start with a buffer (queue) and an empty stack:

```
stack = []
buffer = queue([I, loved, this, movie])
```

Iterate through the transition sequence:

if SHIFT (S): take **first** word (*leftmost*) of the **buffer**, push it to the **stack**

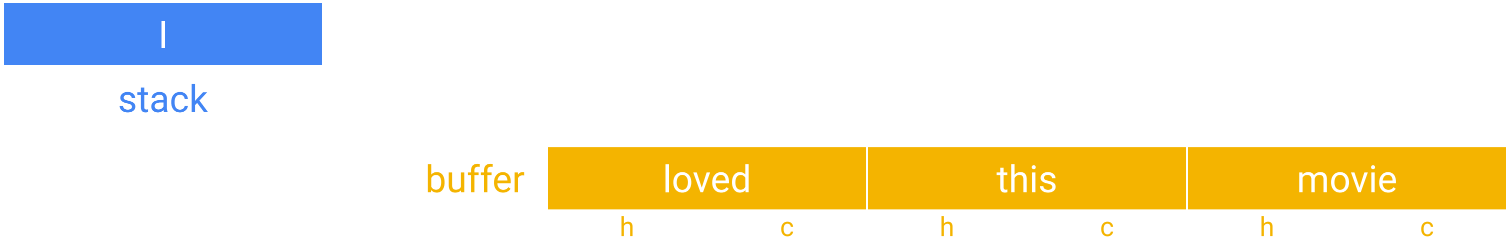
if REDUCE (R): **pop** top 2 words from **stack** + **reduce** them into a **new node (w/ tree LSTM)**

Transition sequence example

```
(I ( loved ( this movie ) ) )
 S   S         S    S      R  R  R
```

Transition sequence example

```
(I ( loved ( this movie ) ) )
 S   S       S    S       R R R
```

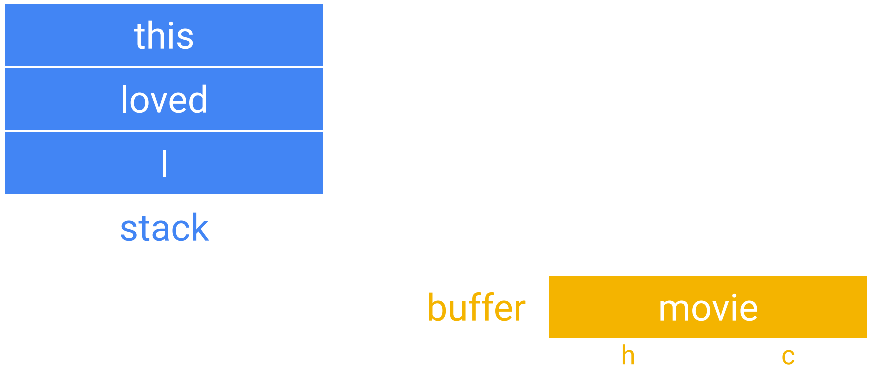


Transition sequence example

```
(I ( loved ( this movie ) ) )
 S   S        S    S     R R R
```

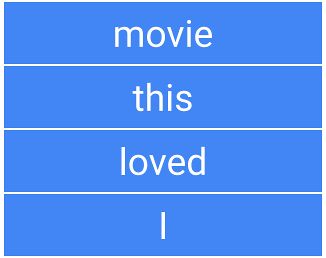
Transition sequence example

```
(I ( loved ( this movie ) ) )
 S   S       S    S      R R R
```



Transition sequence example

```
(I ( loved ( this movie ) ) )
 S   S       S    S     R R R
```

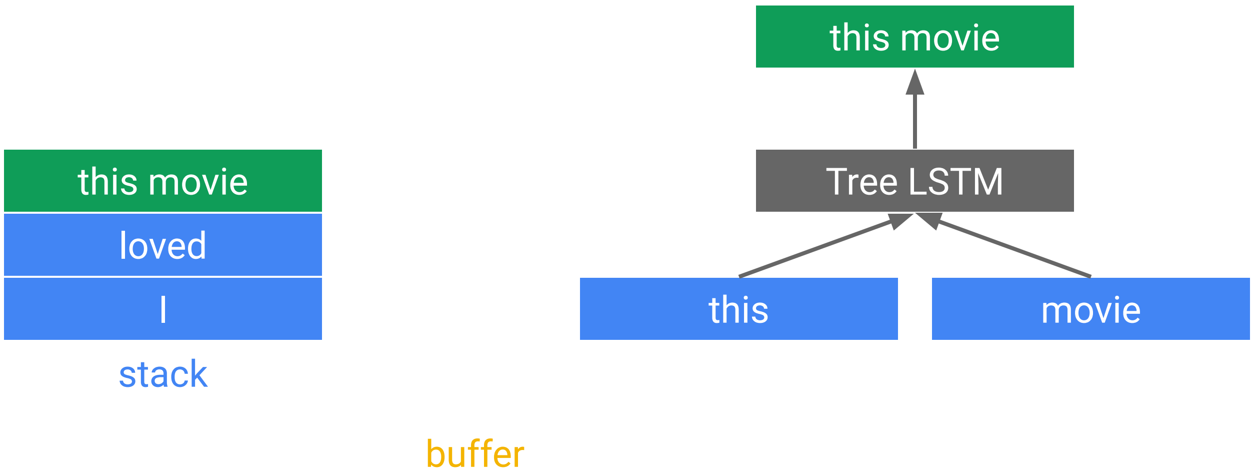


stack

buffer

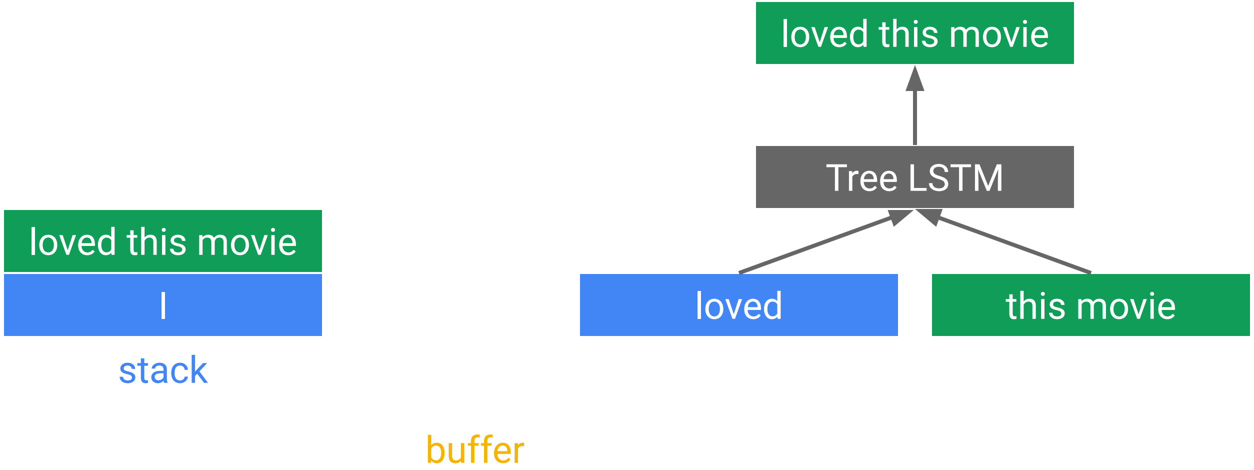
Transition sequence example

```
(I ( loved ( this movie ) ) )
 S   S       S    S     R R R
```



Transition sequence example

```
(I ( loved ( this movie ) ) )
 S   S       S    S     R R R
```



Transition sequence example

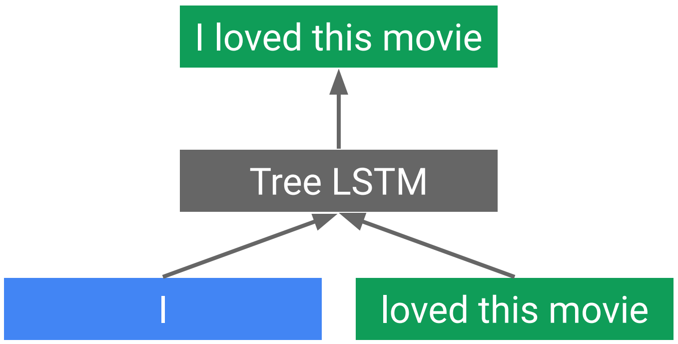
```
(I ( loved ( this movie ) ) )
 S   S       S     S      R R R
```

this is your **root node** for classification

I loved this movie

stack

buffer



Mini-batch SGD

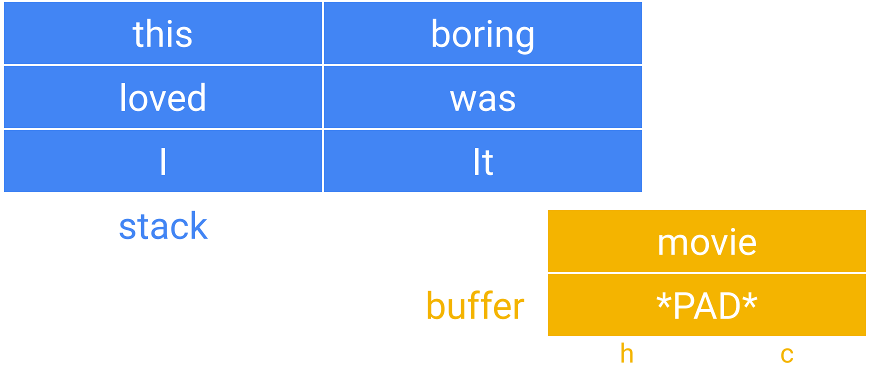
Transition sequence example (mini-batched)

```
(I ( loved ( this movie ) ) )          (It ( was boring ) )
 S   S         S    S       R R R        S     S   S      R R
```

stack	I	loved	this	movie
buffer	It	was	boring	*PAD*
	h c	h c	h c	h c

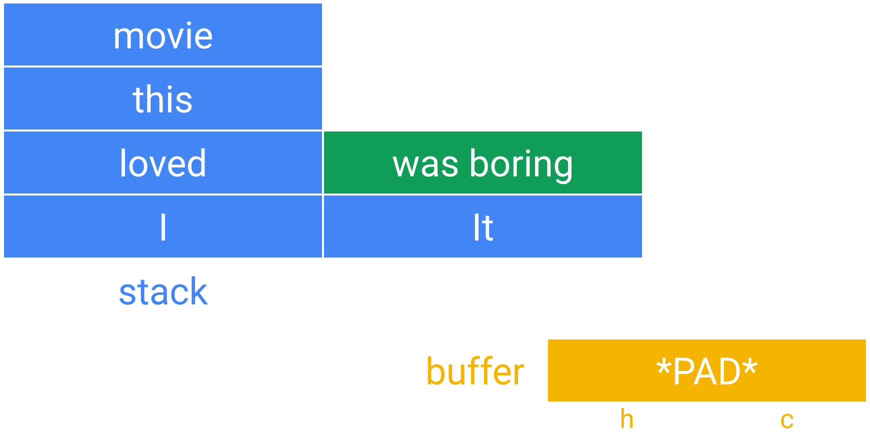
Transition sequence example (mini-batched)

```
(I ( loved ( this movie ) ) )        (It ( was boring ) )
 S   S         S     S     R R R      S     S    S     R R
```

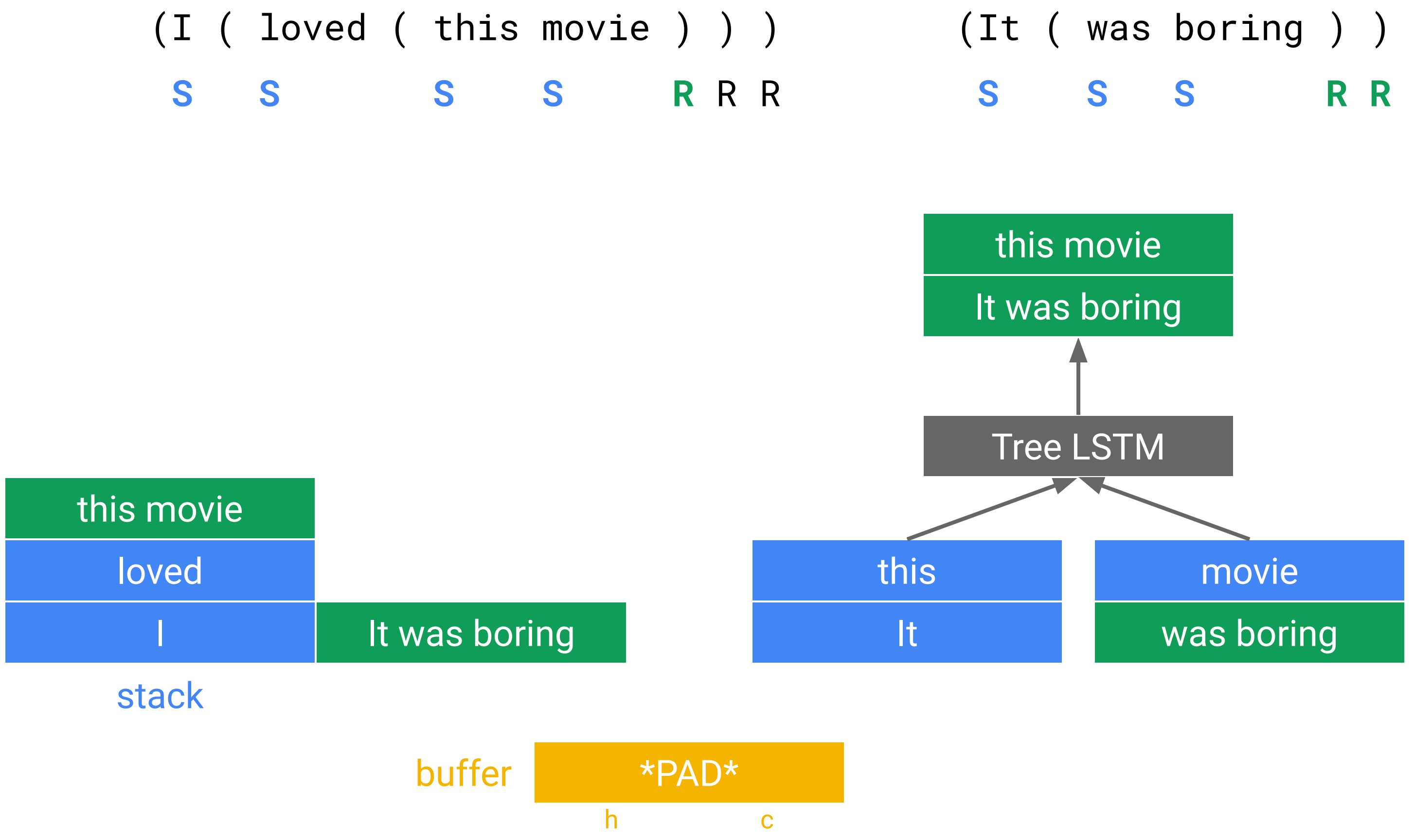


Transition sequence example (mini-batched)

```
(I ( loved ( this movie ) ) )        (It ( was boring ) )
 S   S       S    S     R R R         S    S   S      R R
```

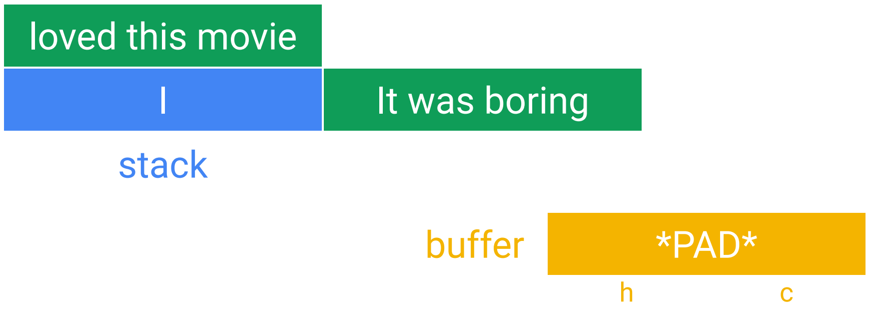


Transition sequence example (mini-batched)



Transition sequence example (mini-batched)

```
(I ( loved ( this movie ) ) )        (It ( was boring ) )
 S   S       S    S     R R R          S   S   S      R R
```



Transition sequence example (mini-batched)

```
(I ( loved ( this movie ) ) )        (It ( was boring ) )
 S   S        S    S      R R R       S    S   S      R R
```

Optional approach: Sentence + Sentiment + Syntax + Node-level sentiment

1. one-sentence review + “global” sentiment score
2. tree structure (syntax)
3. **node-level sentiment scores**

Summary

Recap

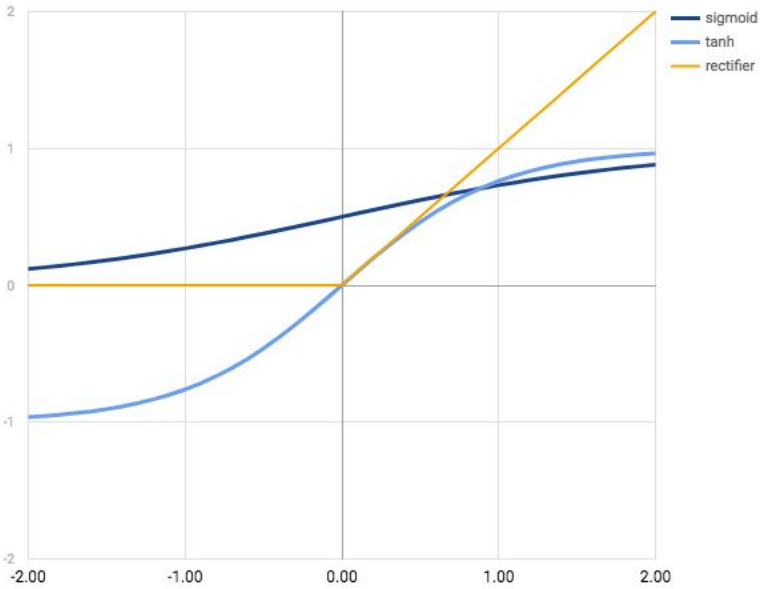
- Training basics
  - SGD
  - Backpropagation
  - Cross Entropy Loss
- Bag of Words models: BOW, CBOW, Deep CBOW
  - Can encode a sentence of arbitrary length, but loses word order
- Sequence models: RNN and LSTM
  - Sensitive to word order
  - RNN has vanishing gradient problem, LSTM deals with this
  - LSTM has input, forget, and output gates that control information flow

Recap

- Tree-based models: Child-Sum & N-ary Tree LSTM
  - Generalize LSTM to tree structures
  - Exploit compositionality, but require a parse tree
  - Transition sequence
- Mini-batch SGD

Extra

Recap: Activation functions



Child-Sum Tree LSTM

useful for encoding **dependency** trees

$$\tilde{h}_j = \sum_{k \in C(j)} h_k,$$

$$i_j = \sigma\left(W^{(i)} x_j + U^{(i)} \tilde{h}_j + b^{(i)}\right),$$

$$f_{jk} = \sigma\left(W^{(f)} x_j + U^{(f)} h_k + b^{(f)}\right),$$

$$o_j = \sigma\left(W^{(o)} x_j + U^{(o)} \tilde{h}_j + b^{(o)}\right),$$

$$u_j = \tanh\left(W^{(u)} x_j + U^{(u)} \tilde{h}_j + b^{(u)}\right)$$

$$c_j = i_j \odot u_j + \sum_{k \in C(j)} f_{jk} \odot c_k,$$

$$h_j = o_j \odot \tanh(c_j),$$