UNIVERSITY OF AMSTERDAM

Lecture 6: Compositional semantics and sentence representations

Rochelle Choenni

NLP1 2022 November 17, 2022

Credits: **Sandro Pezelle, Ekaterina Shutova, J. Bastings, Mario Giulianelli**

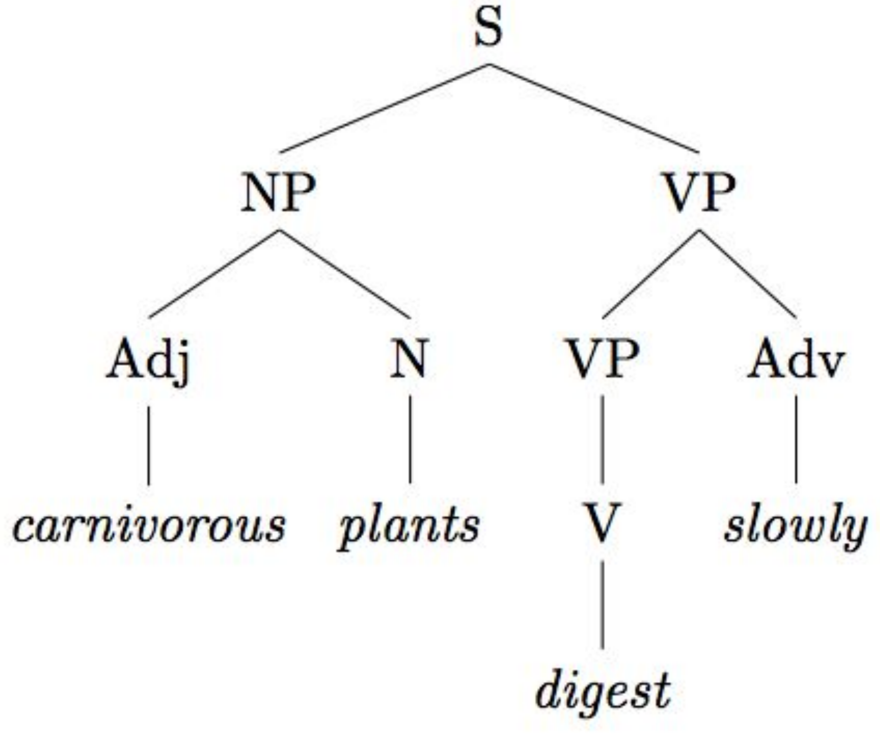
Outline

- Compositional semantics
- Compositional distributional semantics
- Compositional semantics with 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



Non-trivial issues with semantic composition

- Similar syntactic structures may have different meanings
  - ➡ *it barks*
  - ➡ *it rains; it snows* (pleonastic pronoun)

- Different syntactic structures may have the same meaning (e.g., passive constructions)
  - ➡ *Kim ate the apple.*
  - ➡ *The apple was eaten by Kim.*

- 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.

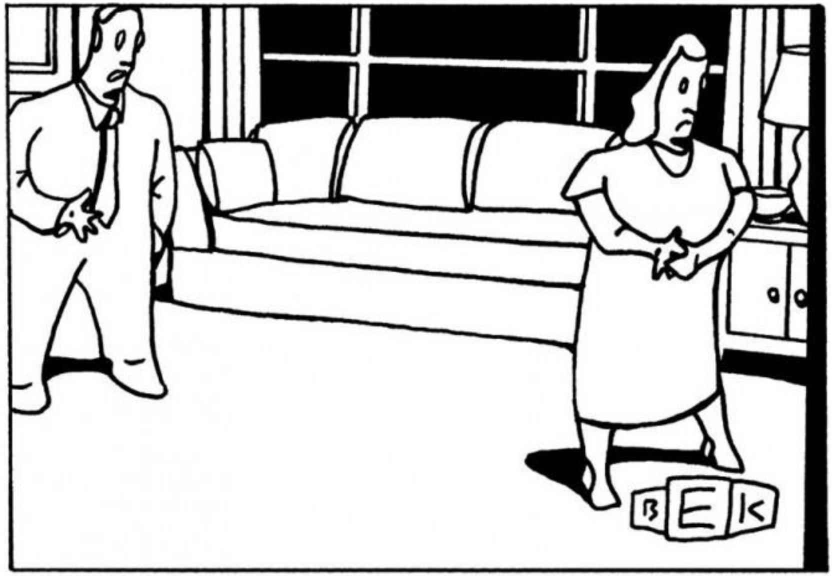
Non-trivial issues with semantic composition

- Additional meaning can arise through composition (e.g., logical metonymy)
  - ➡ *fast programmer*
  - ➡ *fast plane*
  - ➡ *enjoy a book*
  - ➡ *enjoy a cup of tea*

- Meaning transfers and additional connotations can arise through composition (e.g., metaphor)
  - ➡ *I can't **buy** this story.*
  - ➡ *This sum will **buy** you a ride on the train.*

- Recursive composition

Issues with semantic composition



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

Modelling compositional semantics

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

Outline

- Compositional semantics
- Compositional distributional semantics
- Compositional semantics with neural networks

Compositional distributional semantics

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

- Given a finite vocabulary, natural languages licence an infinite amount of sentences.
- So it is impossible to learn vector representations for all sentences.

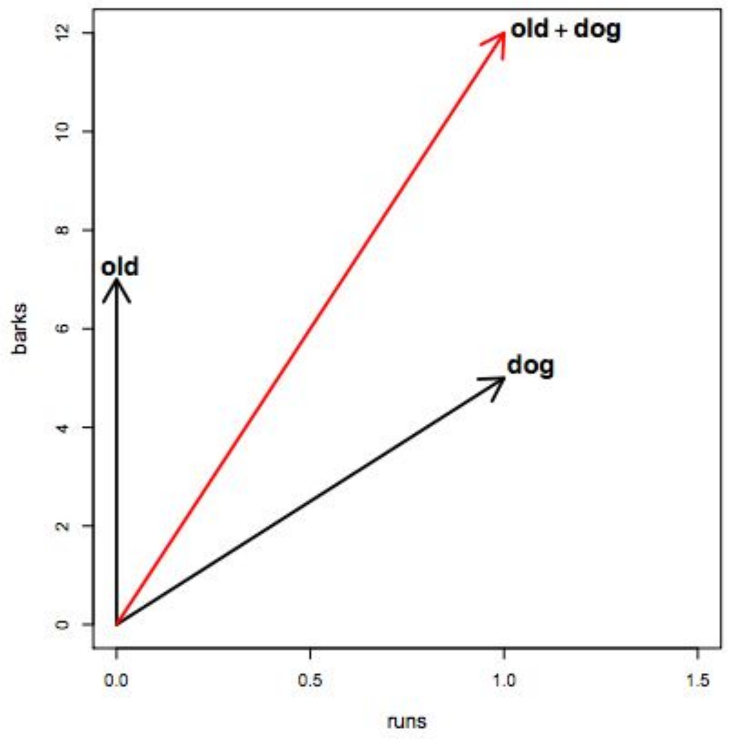
➡ But we can still use distributional word representations and learn to perform **semantic composition in distributional space**.

Vector mixture models

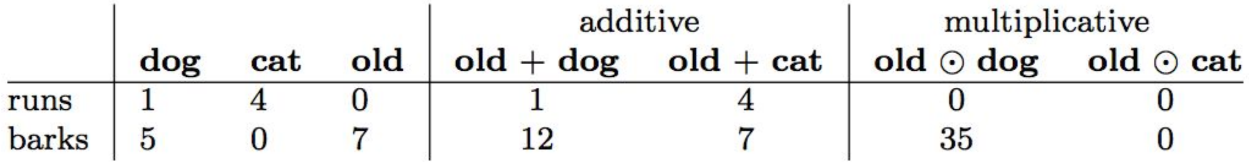
Mitchell and Lapata, 2010. *Composition in Distributional Models of Semantics Models*

- ➔ Additive
- ➔ Multiplicative

Simple, but surprisingly effective!



Additive and multiplicative models



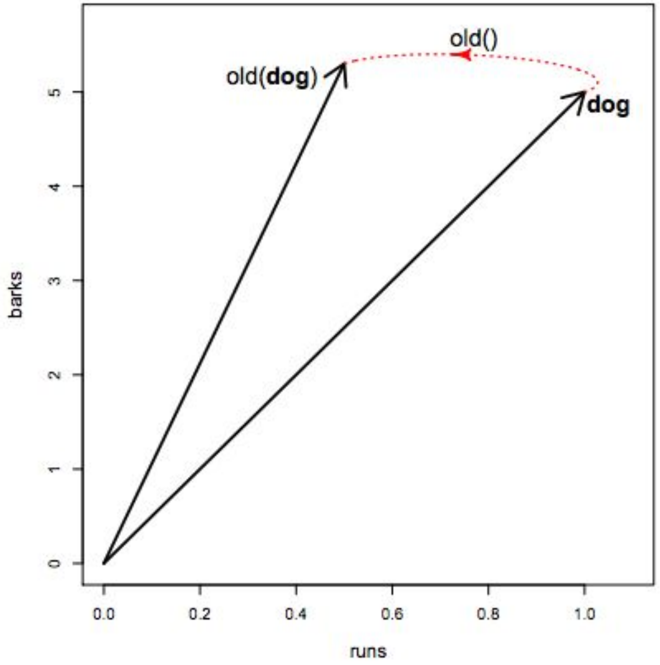
				additive		multiplicative	
	dog	**cat**	**old**	**old + dog**	**old + cat**	**old ⊙ dog**	**old ⊙ cat**
runs	1	4	0	1	4	0	0
barks	5	0	7	12	7	35	0

- Correlate with human similarity judgments about adjective-noun, noun-noun, verb-noun and noun-verb pairs
- The additive and the multiplicative model are **symmetric** (commutative): they do not take word order or syntax into account.
  - ➡ *John hit the ball = The ball hit John*
- More suitable for modelling **content words**, would not apply well to function words (e.g. conjunctions, prepositions etc.):
  - ➡ *<u>some</u> dogs, lice <u>and</u> dogs, lice <u>on</u> dogs*

Lexical function models

Distinguish between:

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



Lexical function models

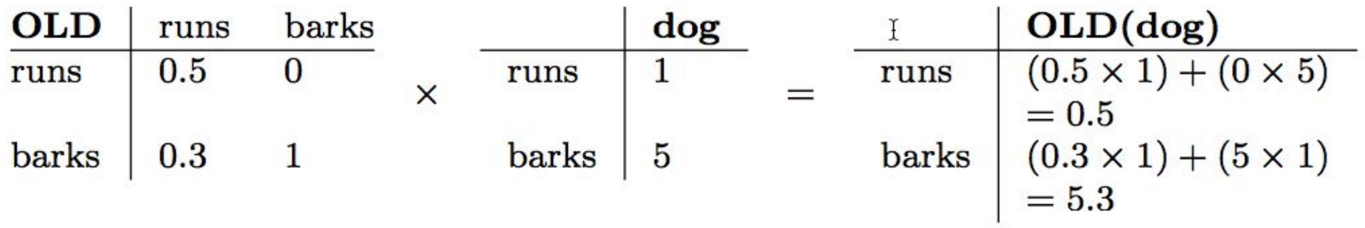
Baroni and Zamparelli. (2010). Nouns are vectors, adjectives are matrices: Representing adjective-noun constructions in semantic space. In *Proceedings of EMNLP.*

Adjectives modelled as lexical functions that are applied to nouns: *old dog* = *old*(*dog*)

➔ Adjectives are parameter matrices ($\mathbf{A}_{old}$, $\mathbf{A}_{furry}$, etc.)

➔ Nouns are vectors (**house**, **dog**, etc.)

➔ Composition is a linear transformation: **old dog** = $\mathbf{A}_{old}$ x **dog**.



OLD	runs	barks
runs	0.5	0
barks	0.3	1

×

	dog
runs	1
barks	5

=

	OLD(dog)
runs	$(0.5 \times 1) + (0 \times 5) = 0.5$
barks	$(0.3 \times 1) + (5 \times 1) = 5.3$

Learning adjective matrices

For each adjective, learn a parameter matrix that allows to predict adjective-noun phrase vectors.

	X	Y
Training set	**house**	**old house**
	dog	**old dog**
	car	**old car**
	cat	**old cat**
	toy	**old toy**
	…	…
Test set	**elephant**	**old elephant**
	mercedes	**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:

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

Outline

- Compositional semantics
- Compositional distributional semantics
- Compositional semantics with neural networks

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**!

Task

Task: Sentiment classification of movie reviews

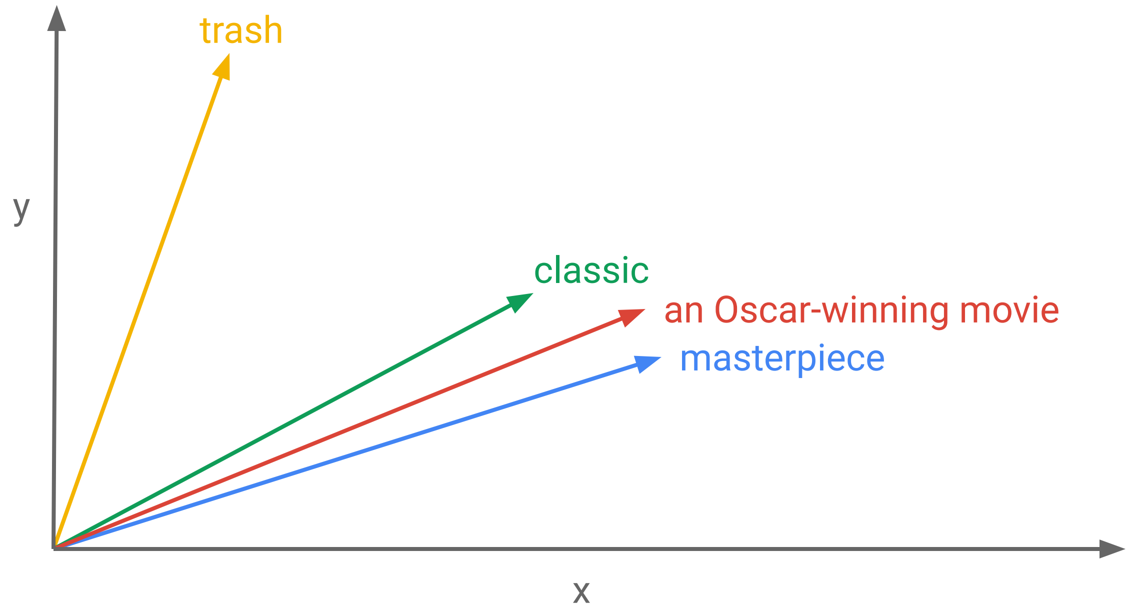
You'll probably love it. →

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

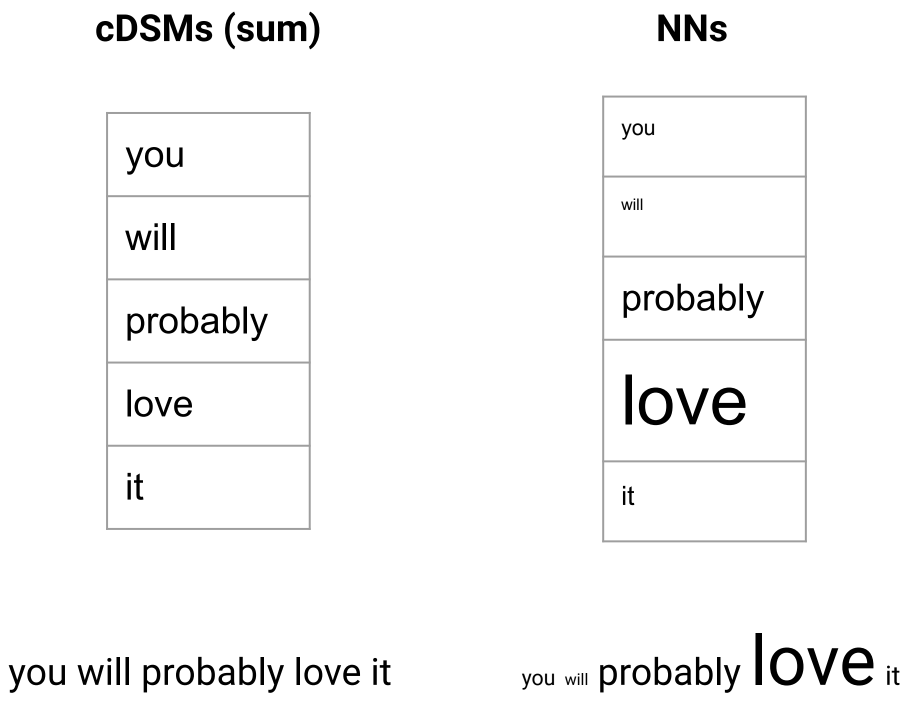
Task-specific: The learned representation has to be "specialized" on **sentiment**!

Words (and sentences) into vectors

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



Sentence representation: A (very) simplified picture



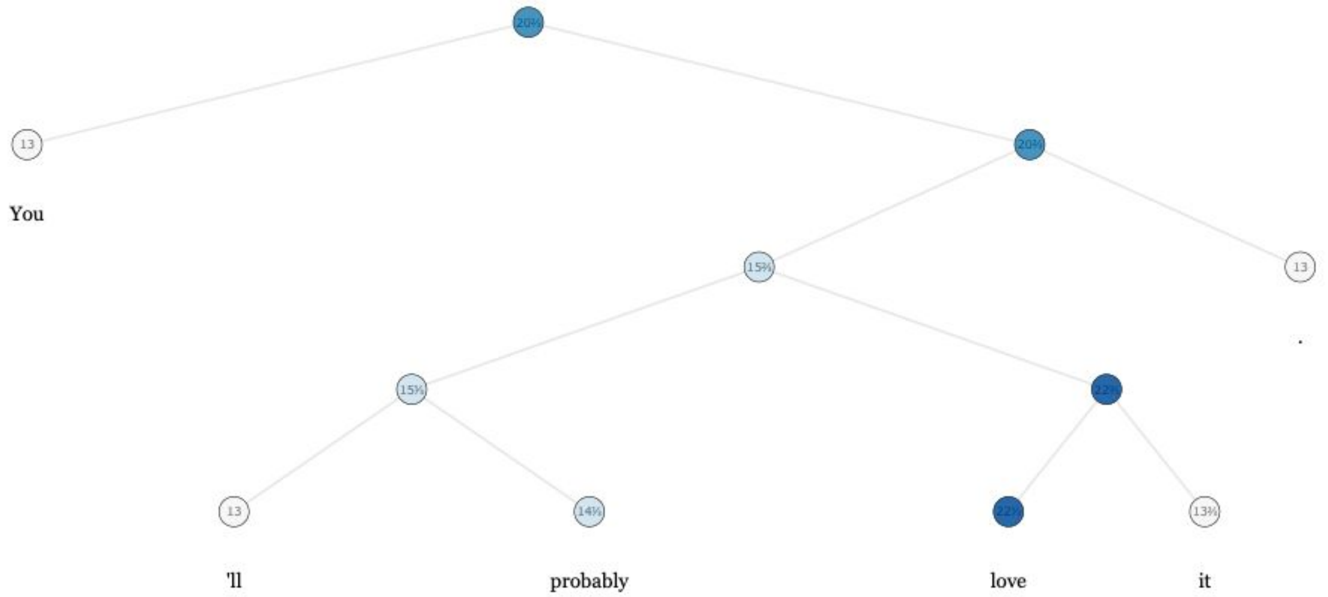
Dataset

Dataset: Stanford Sentiment Treebank (SST)

~12K data-points including:

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

Binary parse tree: One example



Models

Models

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)

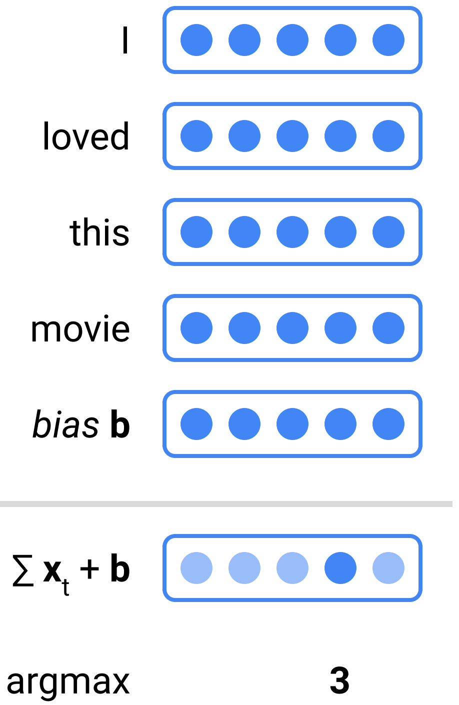
What is a Bag of Words?

- Additive model: does not take word order or syntax into account
- Task-specific word representations with **fixed dimensionality** (*d* = *5*)
- Dimensions of vector space are explicit, **interpretable**

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)
```

I hate that I love this movie = I love that I hate this movie

Turning words into numbers

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

Bad idea: number sequence

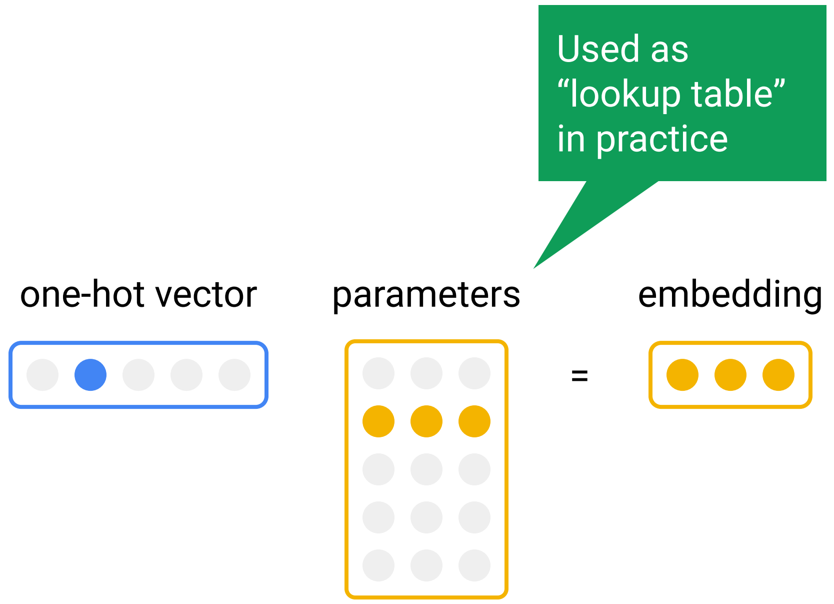
cat	1
tree	2
chair	3
dog	4
mat	5

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



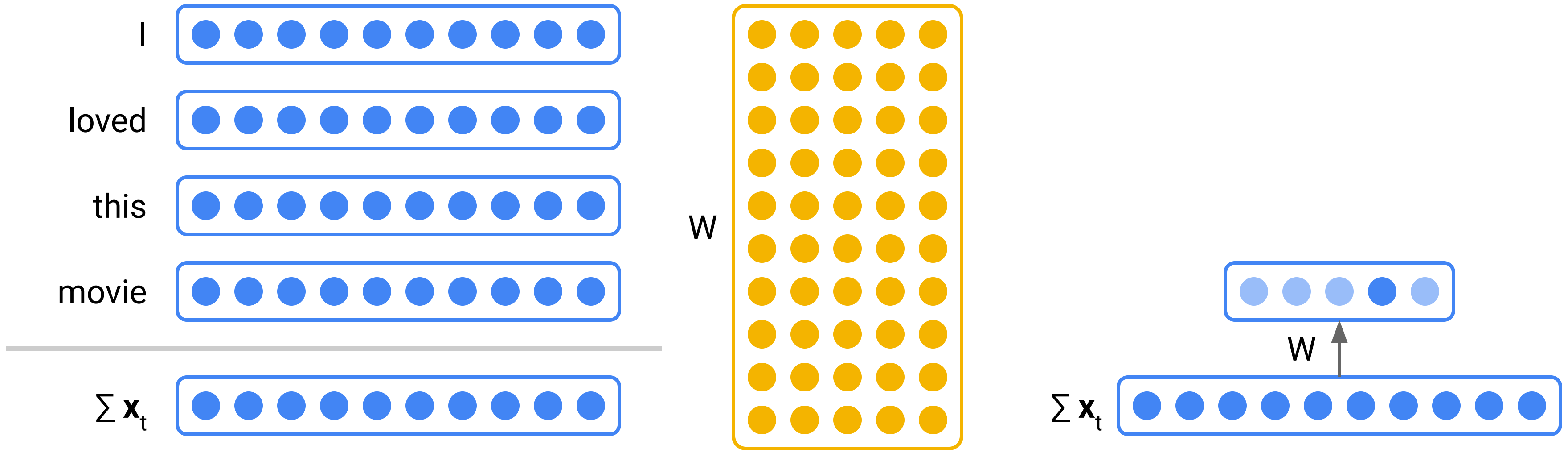
2. Continuous Bag of Words (CBOW)

CBOW

- Additive model: does not take word order or syntax into account
- Task-specific word representations of **arbitrary dimensionality**
- Dimensions of vector space are **not interpretable**
- Prediction can be traced back to the sentence vector dimensions

Continuous Bag of Words (CBOW)

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



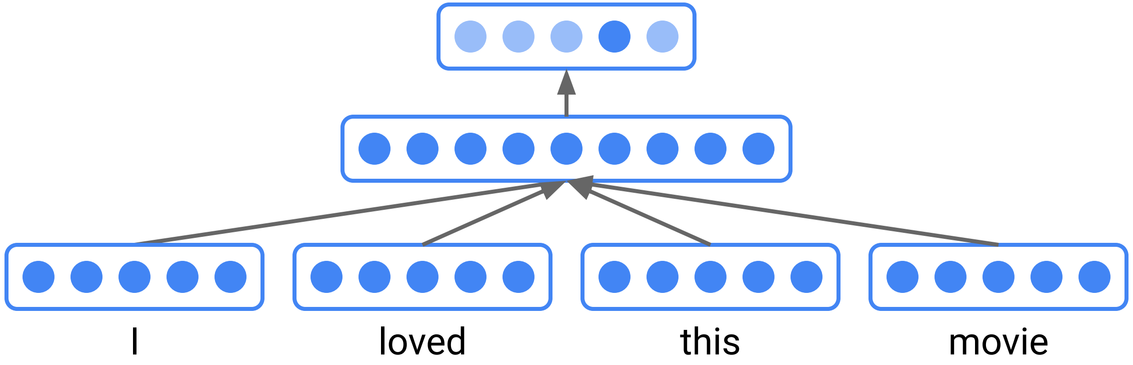
Note that a bias term (of size 5) is added to the final output vector (not shown). Also, this is not the same as word2vec CBOW!

Recall: Matrix Multiplication

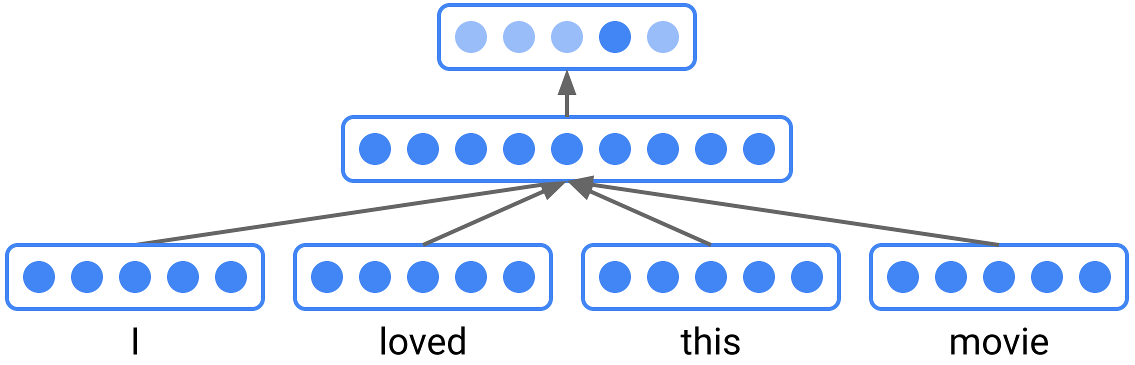
Rows multiply with **columns**

$$\underset{2x3}{\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix}} \times \underset{3x2}{\begin{bmatrix}1 & 2\\ 1 & 2\\ 1 & 2\end{bmatrix}} = \underset{2x2}{\begin{bmatrix}1\times1 + 2\times1 + 3\times1 & 1\times2 + 2\times2 + 3\times2\\ 4\times1 + 5\times1 + 6\times1 & 4\times2 + 5\times2 + 6\times2\end{bmatrix}}$$

What about this?



What about this?



Variable sentence vector size, dependent on sentence length

- Not very sensible conceptually
  - ➡ sentences in a different vector space than words
  - ➡ one vector space for each sentence length in the dataset
- Difficult in practice
  - ➡ what size should the transformation matrix be?
  - ➡ vector size can grow very large

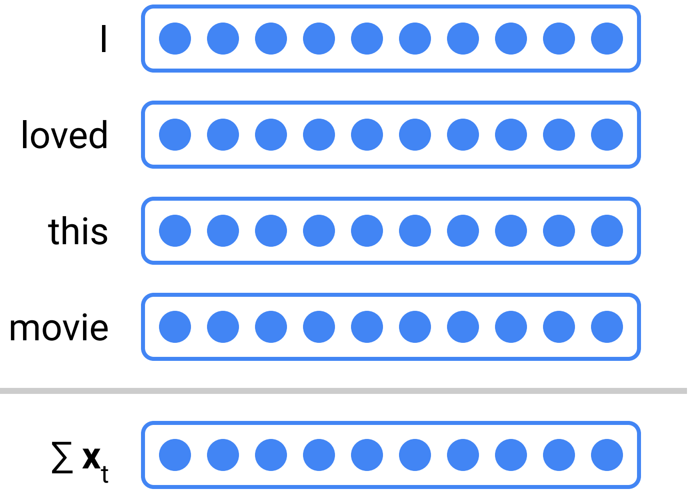
3. Deep CBOW

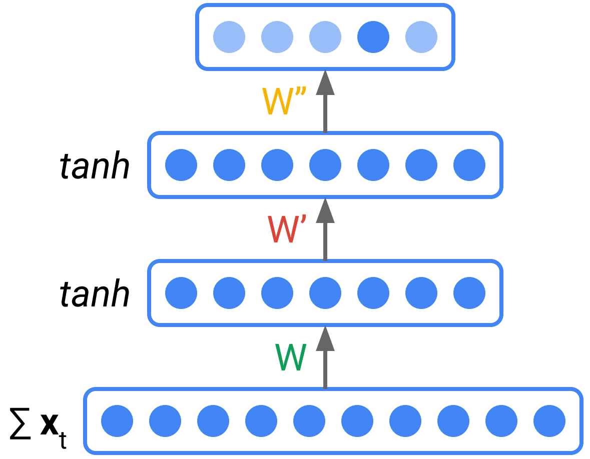
Deep CBOW

- Additive model: does not take word order or syntax into account
- Task-specific word representations of **arbitrary dimensionality**
- Dimensions of vector space are **not interpretable**
- **More layers and non-linear transformations**: prediction cannot be easily traced back

Deep CBOW

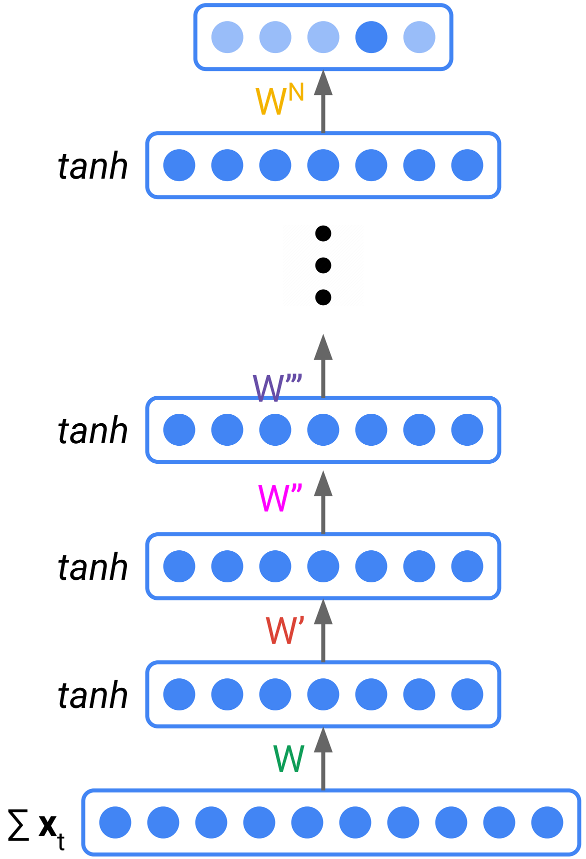
$$W'' \tanh\Big(W' \tanh(W (\textstyle\sum \mathbf{x}_t) + \mathbf{b}) + \mathbf{b}' \Big) + \mathbf{b}''$$





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

What about this?



Is more complexity always better?

Question

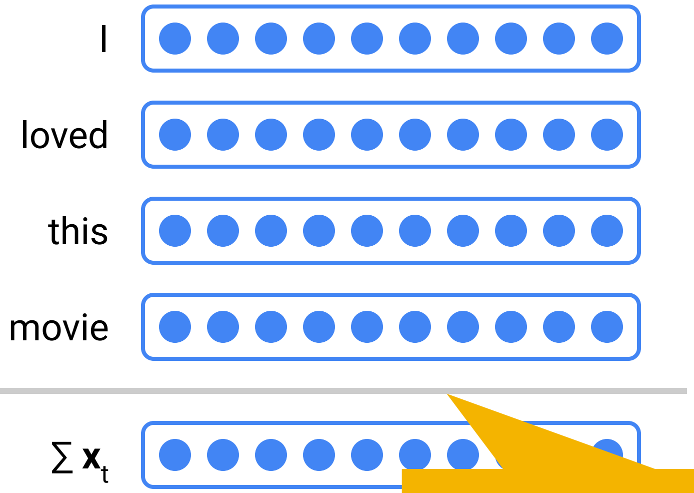
We can learn more complex features, but the only error signal that we receive comes from sentiment prediction.

How can we further help the model?

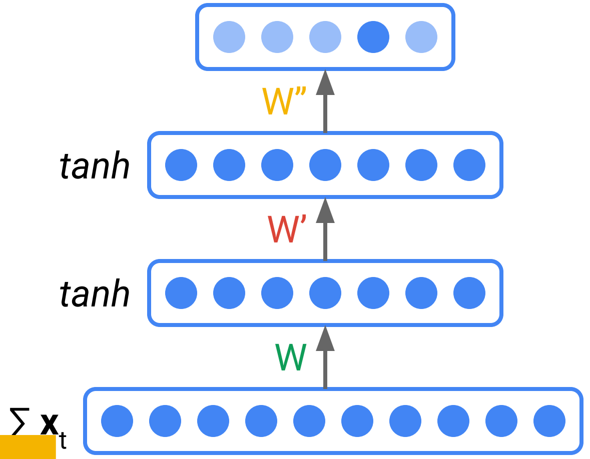
4. Deep CBOW + Pretrained embeddings

Deep CBOW with pretrained embeddings

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



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 + pre-trained embeddings

- Additive model: does not take word order or syntax into account
- Dimensions of vector space are not interpretable
- Multiple layers and non-linear transformations: prediction cannot be easily traced back
- Pre-trained **general-purpose** word representations (e.g., Skip-gram, GloVe)
  - ➡ **keep frozen**: not updated during training
  - ➡ **fine-tune**: updated with task-specific learning signal (*specialised*)

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**

o = [-0.1, 0.1, 0.1, **2.4**, 0.2]

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

softmax$(o_i) = \exp(o_i) / \sum_j \exp(o_j)$

This makes **o** sum to 1.0:

softmax(**o**) = [0.0589, 0.0720, 0.0720, **0.7177**, 0.0795]

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)

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

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:

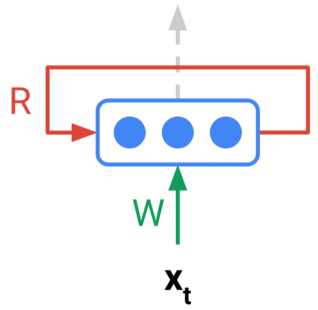
$\mathbf{h}_t = f(\,\mathbf{x}_t, \mathbf{h}_{t-1}\,)$

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

the -> $\mathbf{h}_1 = f(\mathbf{x}_1,\ \mathbf{h}_0)$
cat -> $\mathbf{h}_2 = f(\mathbf{x}_2,\ \mathbf{h}_1)$
sat -> $\mathbf{h}_3 = f(\mathbf{x}_3,\ \mathbf{h}_2)$
…
mat -> $\mathbf{h}_6 = f(\mathbf{x}_6,\ \mathbf{h}_5)$

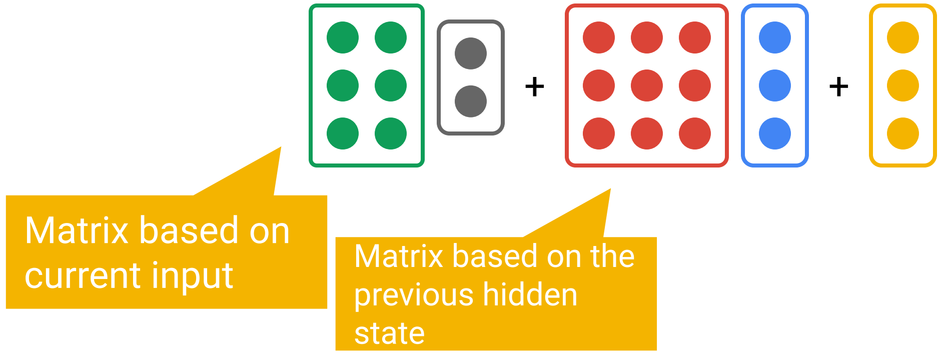
Introduction: Recurrent Neural Network (RNN)

The transition function f consists of an affine transformation followed by a non-linear activation

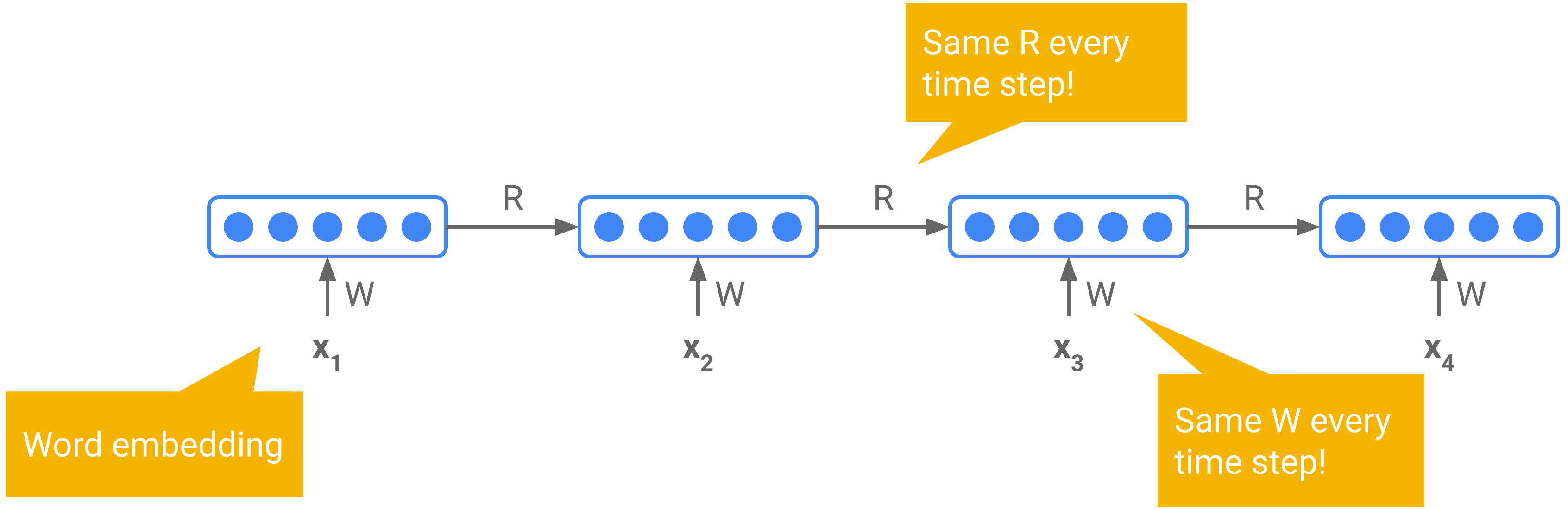


$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1})$$

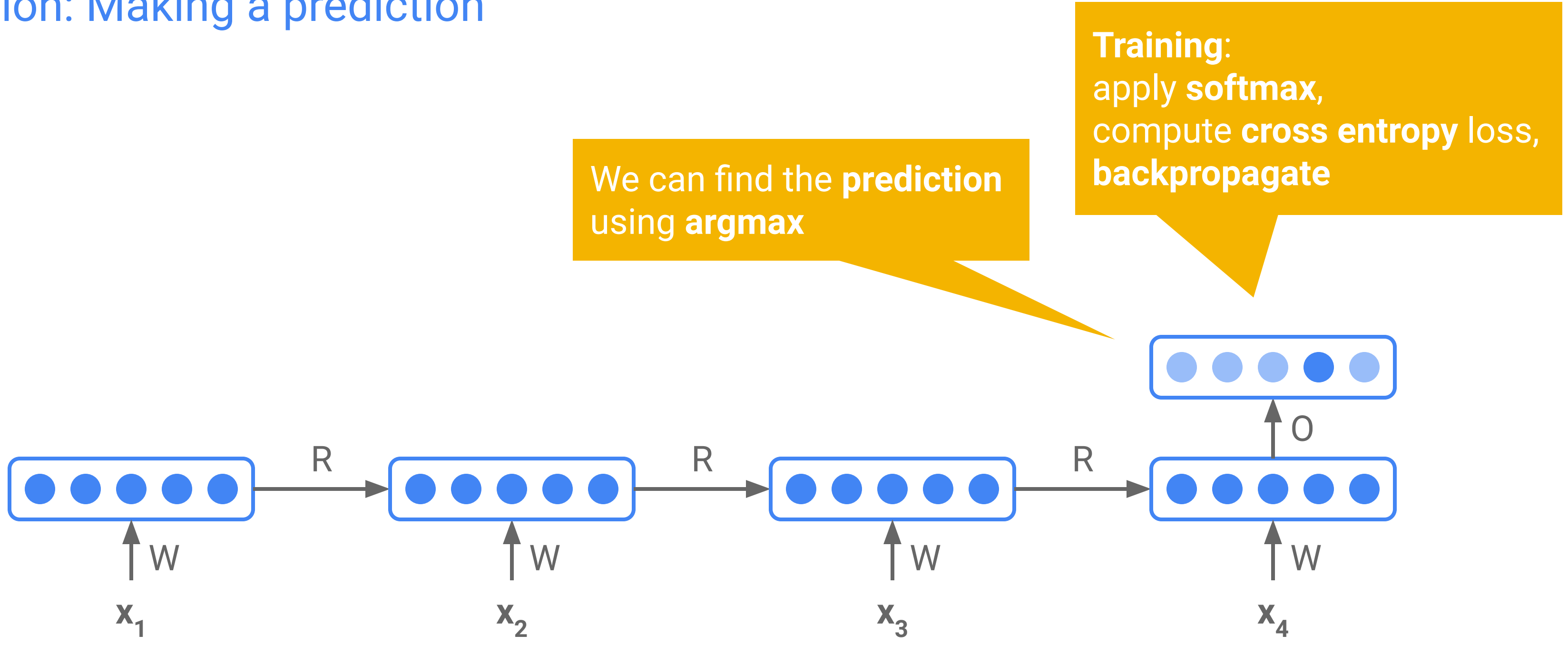
$$= \sigma(W\mathbf{x}_t + R\mathbf{h}_{t-1} + \mathbf{b})$$


+
+
Matrix based on current input
Matrix based on the previous hidden state

Introduction: Unfolding the RNN



Introduction: Making a prediction

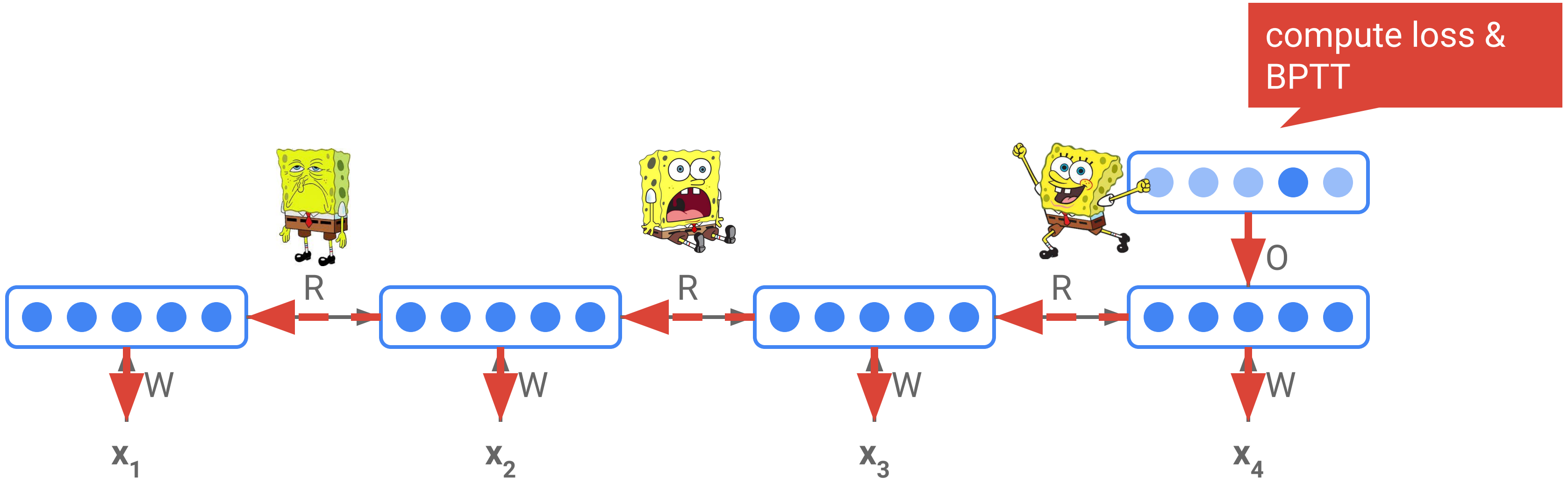


Introduction: The vanishing gradient problem

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.

Introduction: The vanishing gradient problem

R is shared across every timestep!

Imagine that R contains an entry value $r_1 = 0.5$

The first input gets multiplied by $\mathbf{0.5^{\text{num. unrolls N}}}$

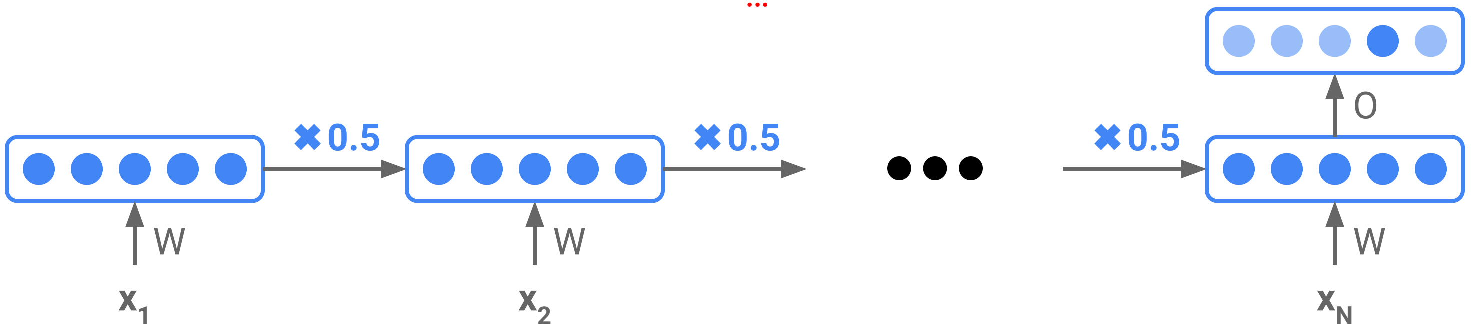
$0.5^5 \sim 0.03$

$0.5^{10} \sim 9\text{e-}4$

$0.5^{15} \sim 3\text{e-}5$

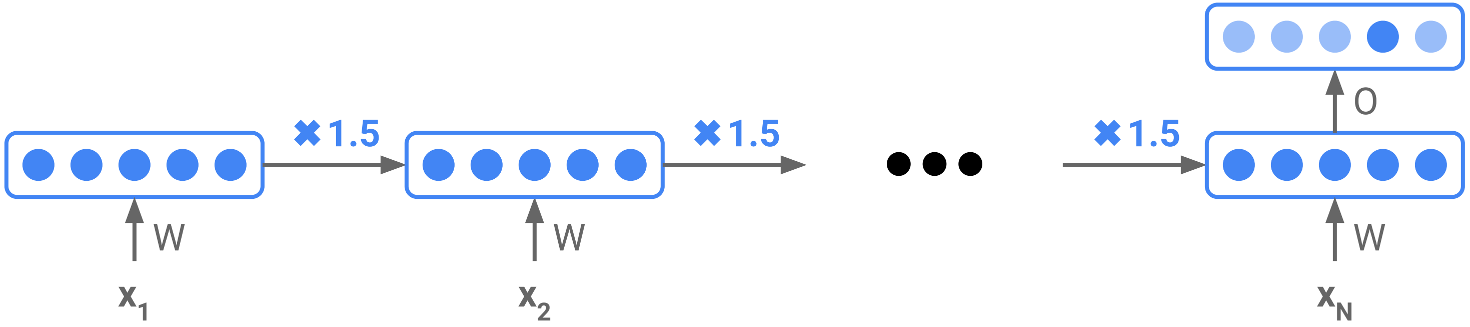
$0.5^{20} \sim 9\text{e-}7$

...



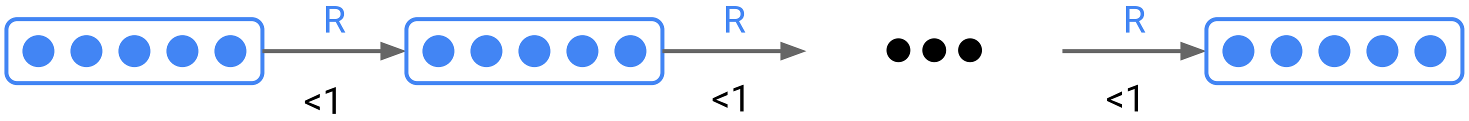
For more details see: Kyunghyun Cho. Natural Language Understanding with Distributed Representation. Section 4.3.

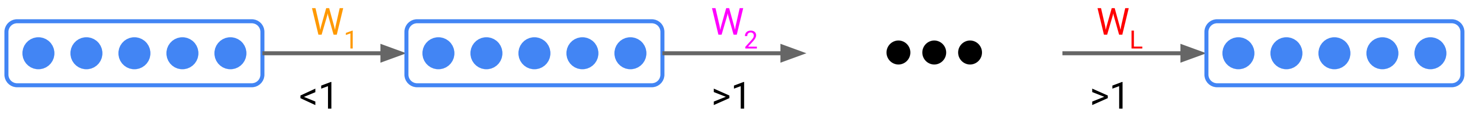
What about this?



Similar problem called exploding gradients!

RNN vs ANN





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 by alleviating the vanishing gradient problem in RNNs

“ I lived in **France** for a while when I was a kid so I can speak fluent…” -> French

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735-1780.

LSTM: Core idea

1. Maintain a **separate memory cell state c_t** from what is outputted (long term memory)
2. Use gates to control the flow of information:
   a. **Forget** gate gets rid of irrelevant information
   b. Input gate to **store** new relevant information from the current input
   c. Selectively **update** the cell state
   d. **Output** gate returns a filtered version of the cell state
3. Backpropagation through time with partially **uninterrupted gradient flow**

LSTMs

RNN:

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

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

LSTM:

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

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

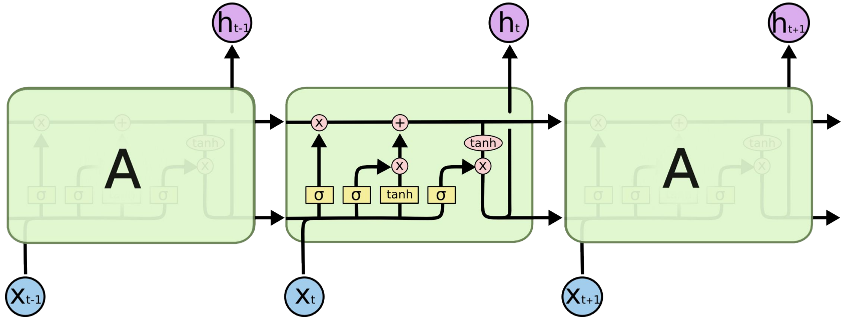
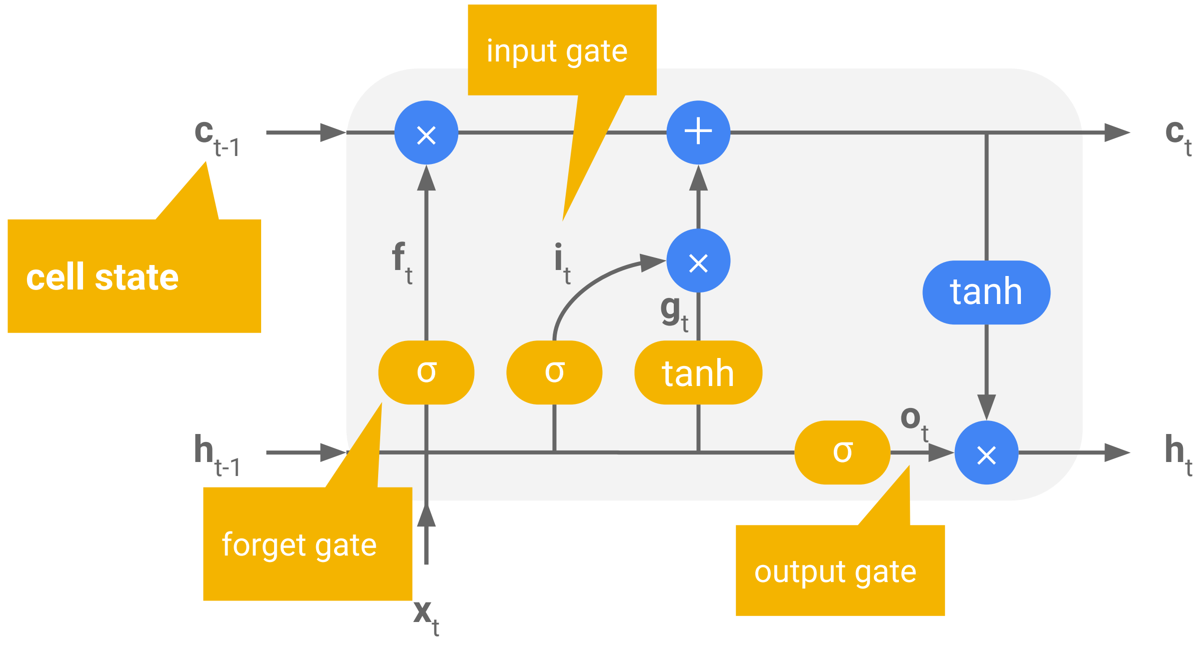


Image credits: http://colah.github.io/posts/2015-08-Understanding-LSTMs

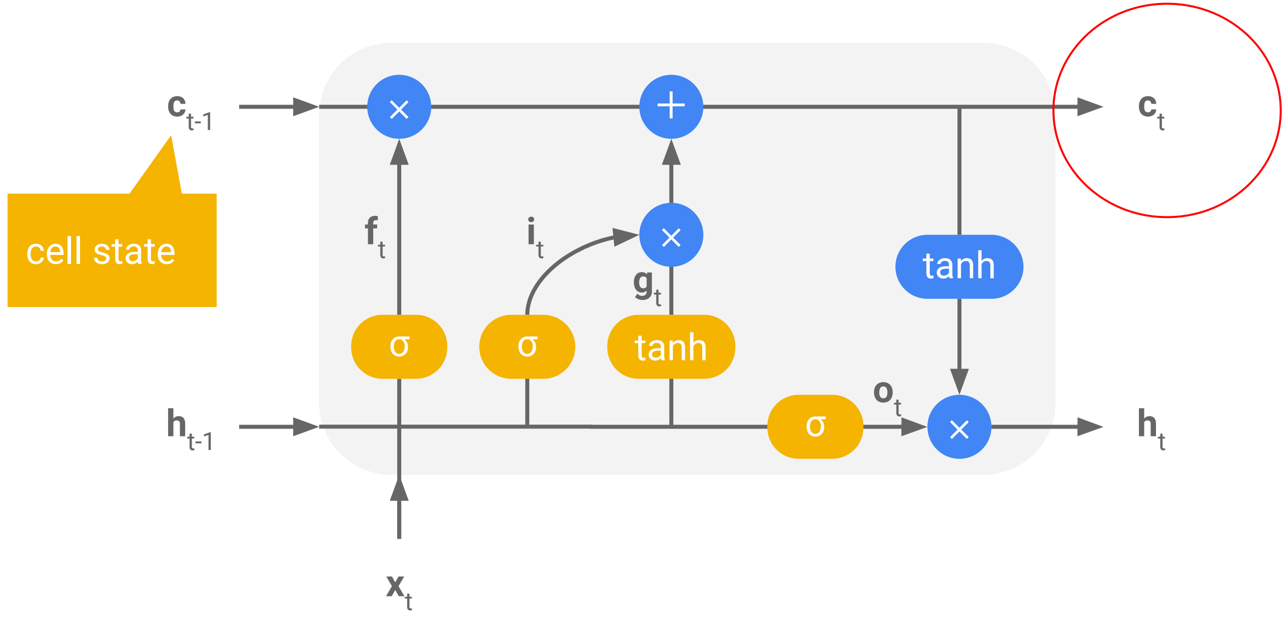
LSTM cell



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

LSTM: Cell state

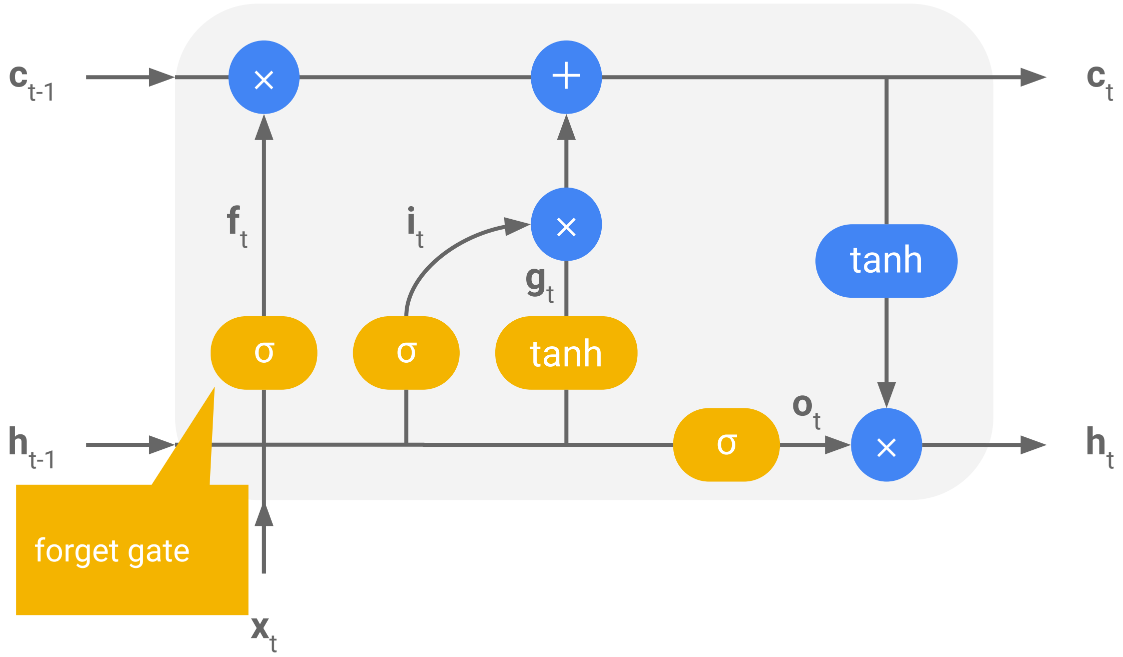
Runs straight down the entire chain, with only some minor linear interactions. LSTM can remove or add information to the cell state, carefully regulated by structures called gates.



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

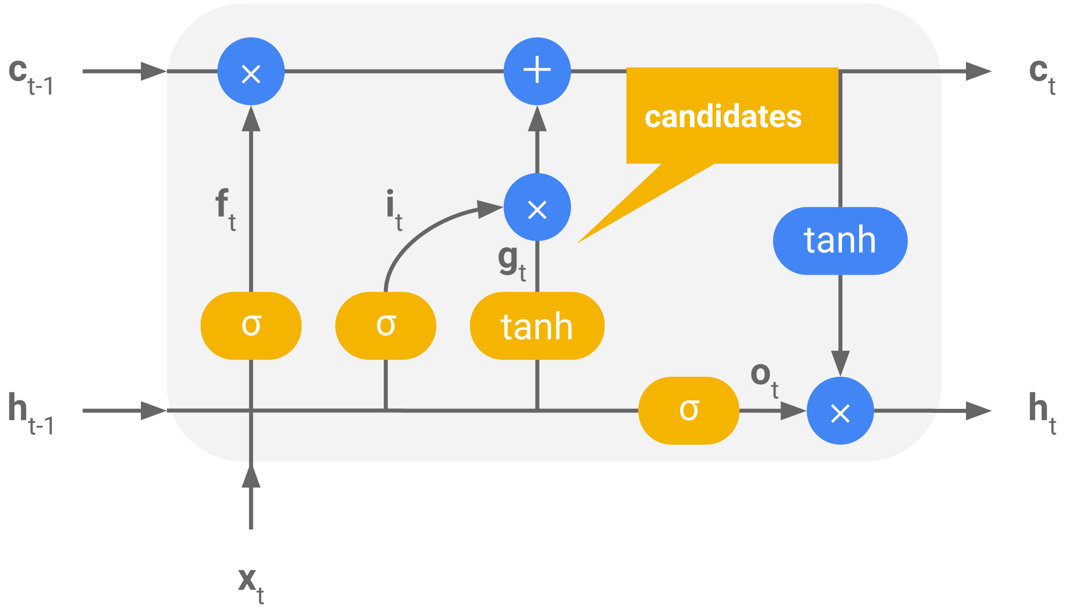
LSTM: Forget gate

Decide what information to throw away from the cell state.



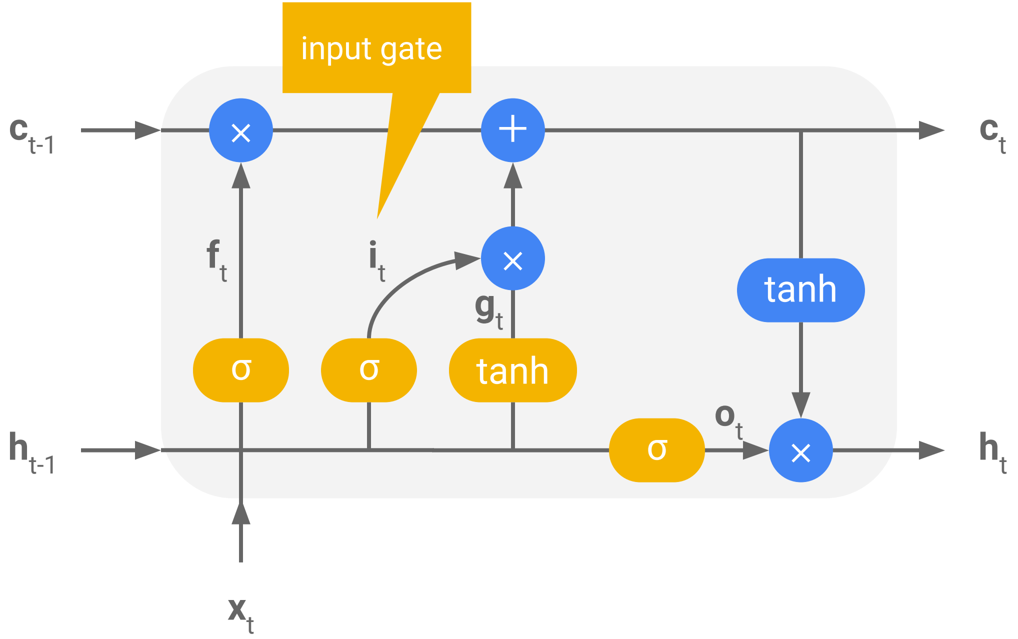
LSTM: Candidate cell

Extracts new **candidate values**, $\mathbf{g}_t$, from the previous hidden state and the current input that could be added to the cell state.



LSTM: Input gate

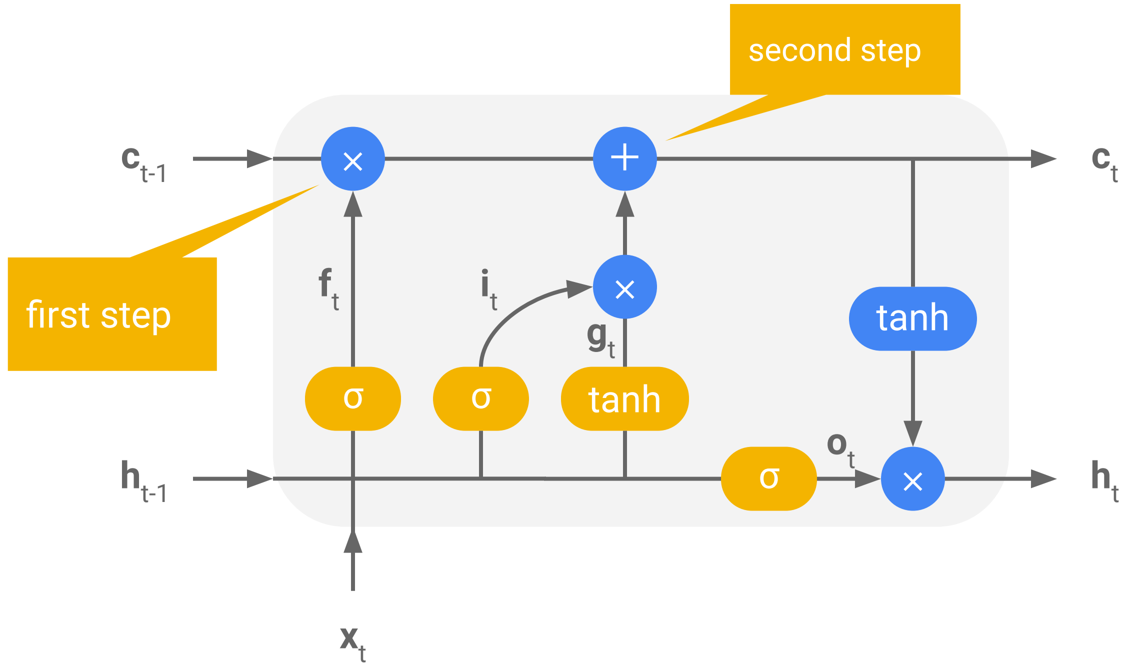
Decide what new information to store in the cell state.



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

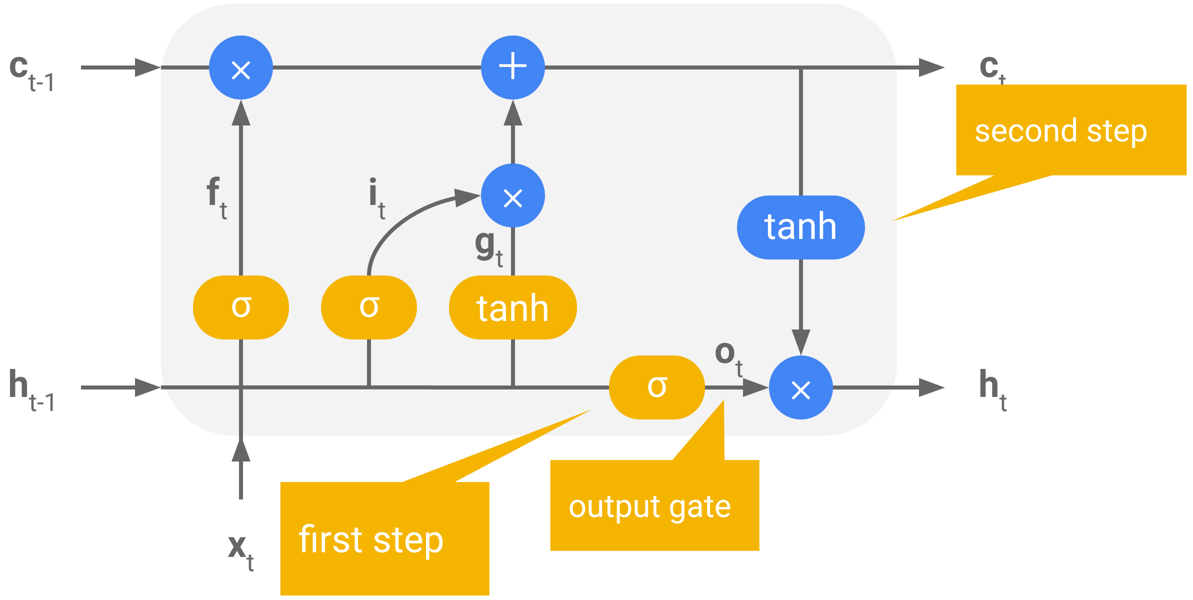
LSTM

Update the cell state: 1. forget things we decided to forget earlier, 2. add the new candidate values scaled by how much we decided to update each state value



LSTM: Output gate

1. Decide what parts of the cell state we're going to output, 2. the cell state is put through *tanh* and multiplied by the output of the output gate, so that we only output the parts we decided to.



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

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

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!

6. Tree LSTM

Sentence representations with NNs

- **Bag of Words models**
  - ➡ sentence representations are **order-independent** function of the word representations
- **Sequence models**
  - ➡ sentence representations are an **order-sensitive** function of a sequence of word representations (surface form)
- **Tree-structured models**
  - ➡ sentence representations are a function of the word representations, **sensitive to the syntactic structure** of the sentence

Second approach: Sentence + Sentiment + Syntax

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

Exploiting tree structure

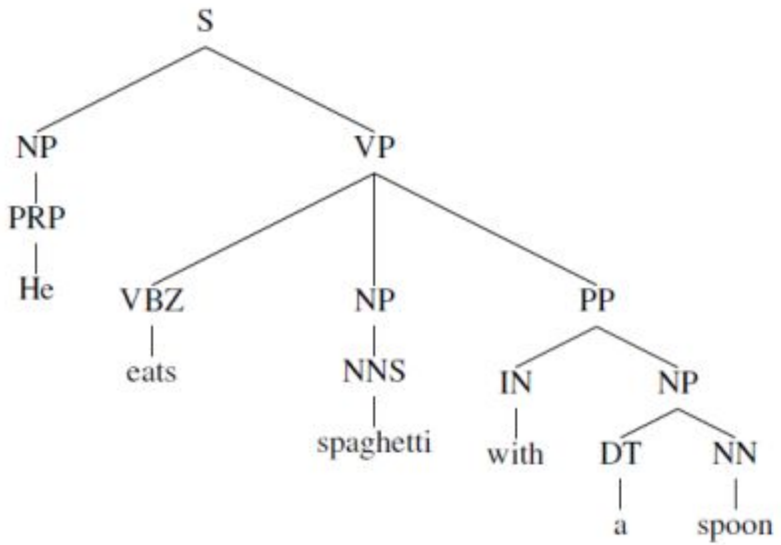
Instead of treating our input as a **sequence**, we can take an alternative approach: assume a **tree structure** and use the principle of **compositionality**.

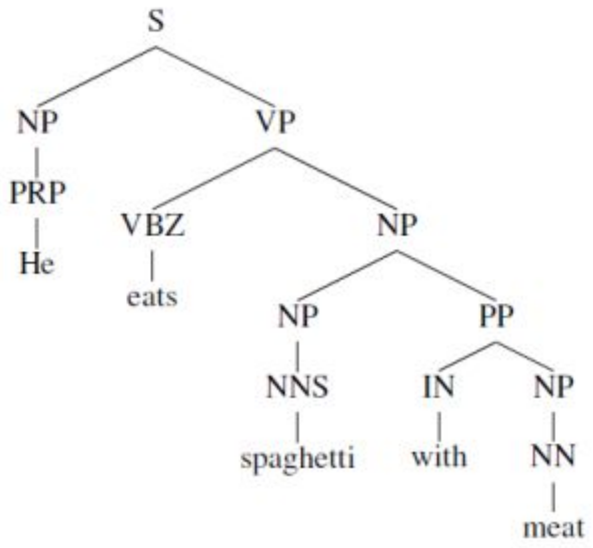
The meaning (vector) of a sentence is determined by:

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

Why would it be useful?

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

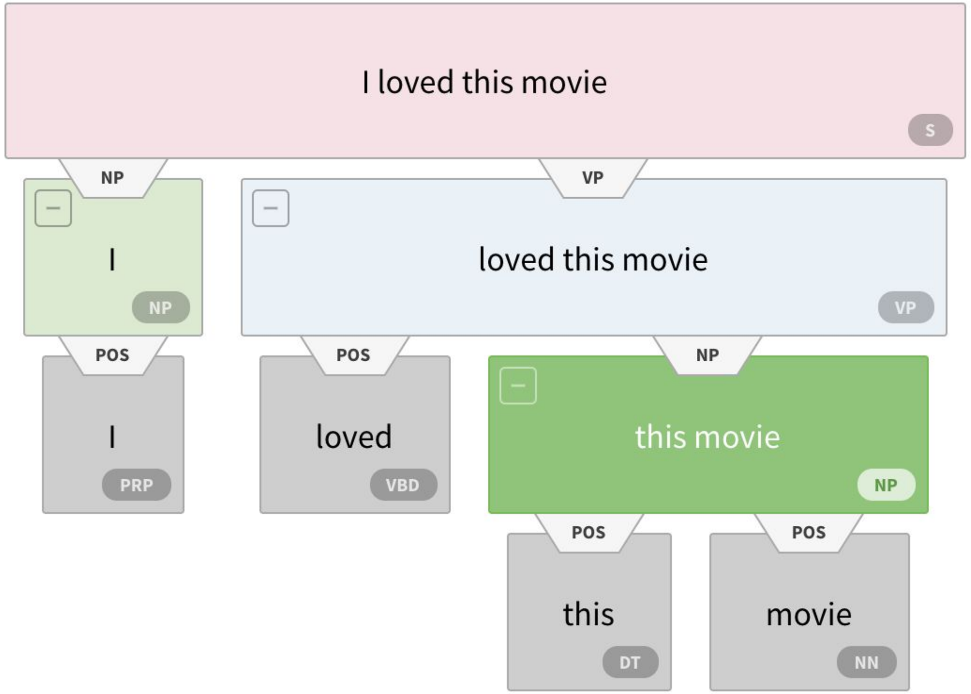




Credits: https://cs224d.stanford.edu/lectures/CS224d-Lecture10.pdf

Constituency Parse

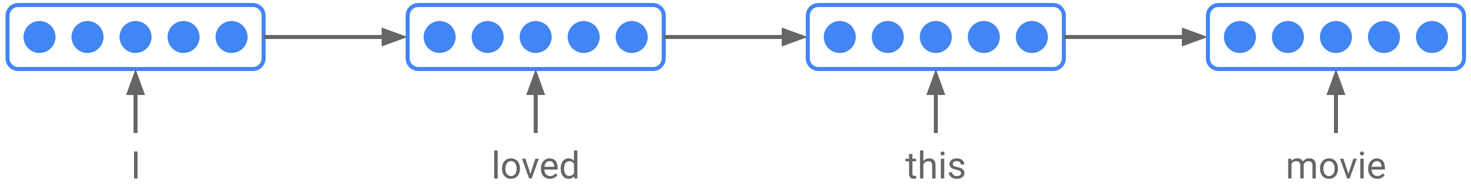
Can we obtain a **sentence vector** using the tree structure given by a parse?



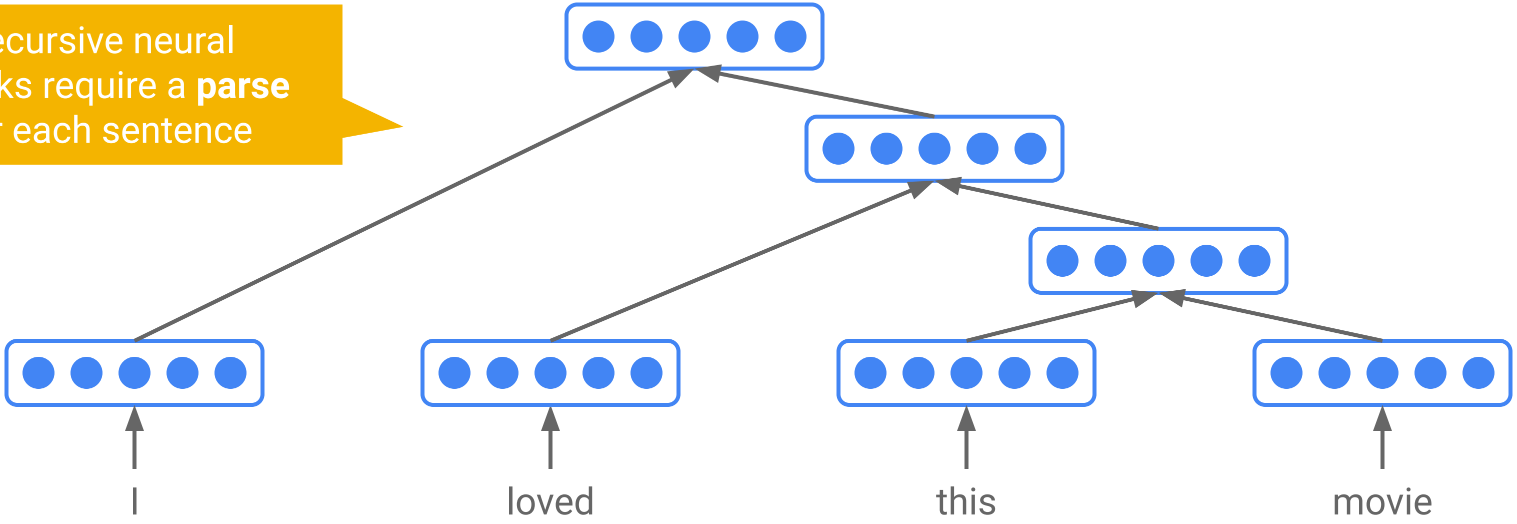
http://demo.allennlp.org/constituency-parsing

Recurrent vs Tree Recursive NN

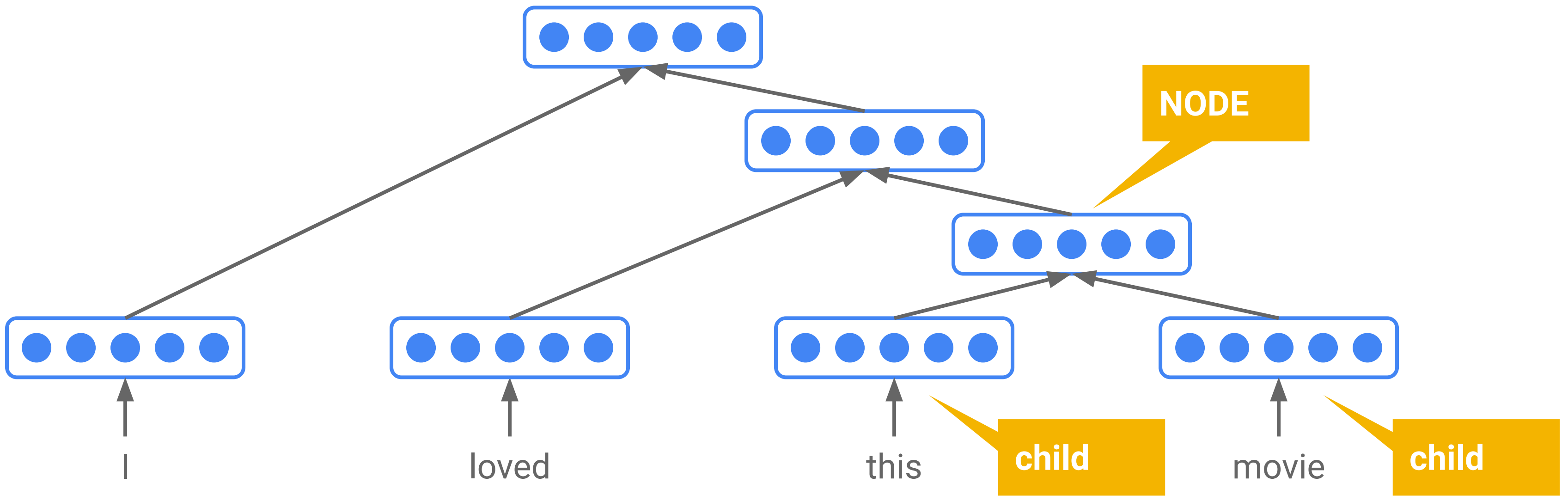
RNNs cannot capture phrases **without prefix context** and often capture too much of **last words** in final vector



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



Tree Recursive NN



Adapted from Stanford cs224n.

Practical II data set: Stanford Sentiment Treebank (SST)

sentiment label for **root node**

```
             3
 ------------|--------------------
|                                 4
|        -------------------------|----------------------------------------------
|       4                                                                        |
|    ---|--------------                                                          |
|   |                 4                                                          |
|   |         ---------|----------                                               |
|   |        |                    3                                              |
|   |        |               -----|----------------------                        |
|   |        |              |                            4                       |
|   |        |              |            ----------------|-------                |
|   |        |              |           |                        2               |
|   |        |              |           |                 -------|---            |
|   |        3              |           |                |           2           |
|   |    ----|-----         |           |                |        ---|-----      |
|   |   |         4         |           3                |       2         |     |
|   |   |     -----|---     |      -----|-------         |    ---|---      |     |
2   2   2    3         2    2     3             2        2   2       2     2     2
|   |   |    |         |    |     |             |        |   |       |     |     |
It  's  a  lovely     film with lovely     performances  by Buy     and Accorsi  .
```

sentiment label for **each node**

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

1. 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

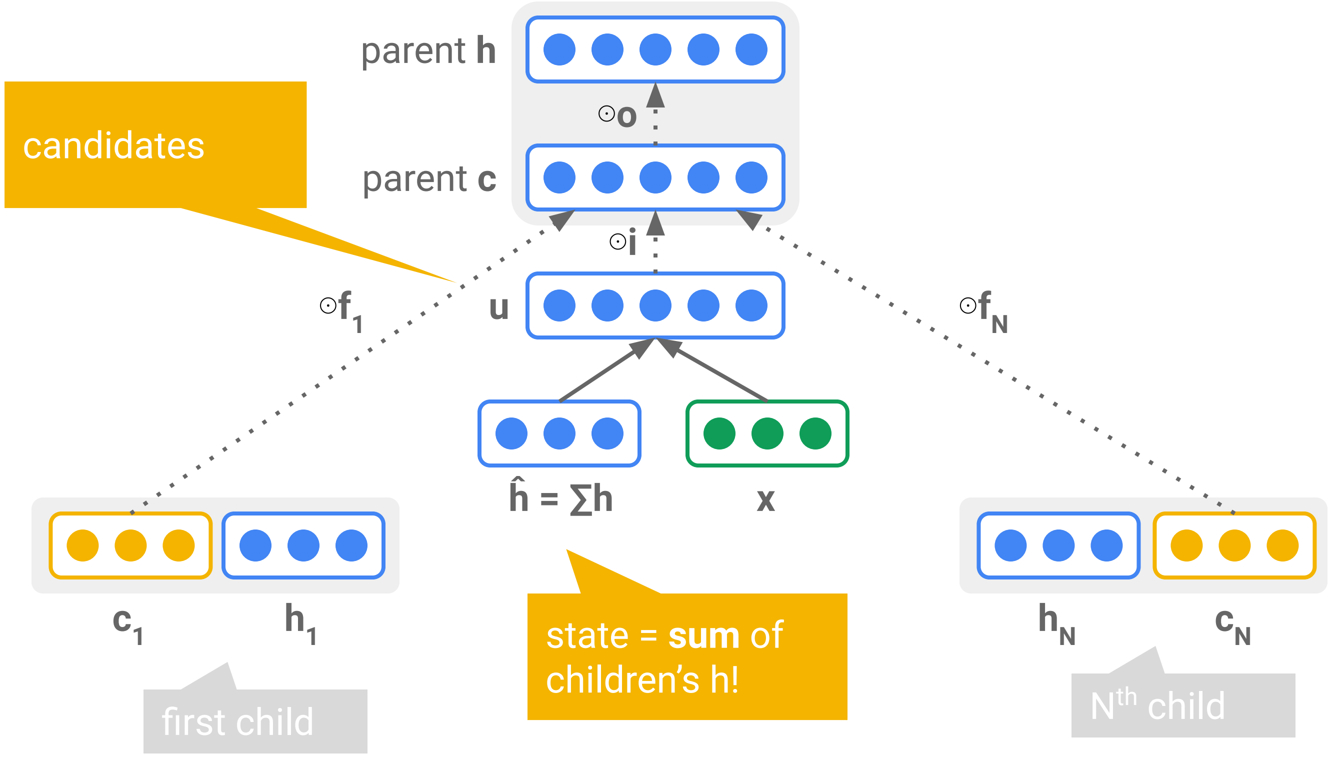
Credits: Daniel Perez https://www.slideshare.net/tuvistavie/tree-lstm

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



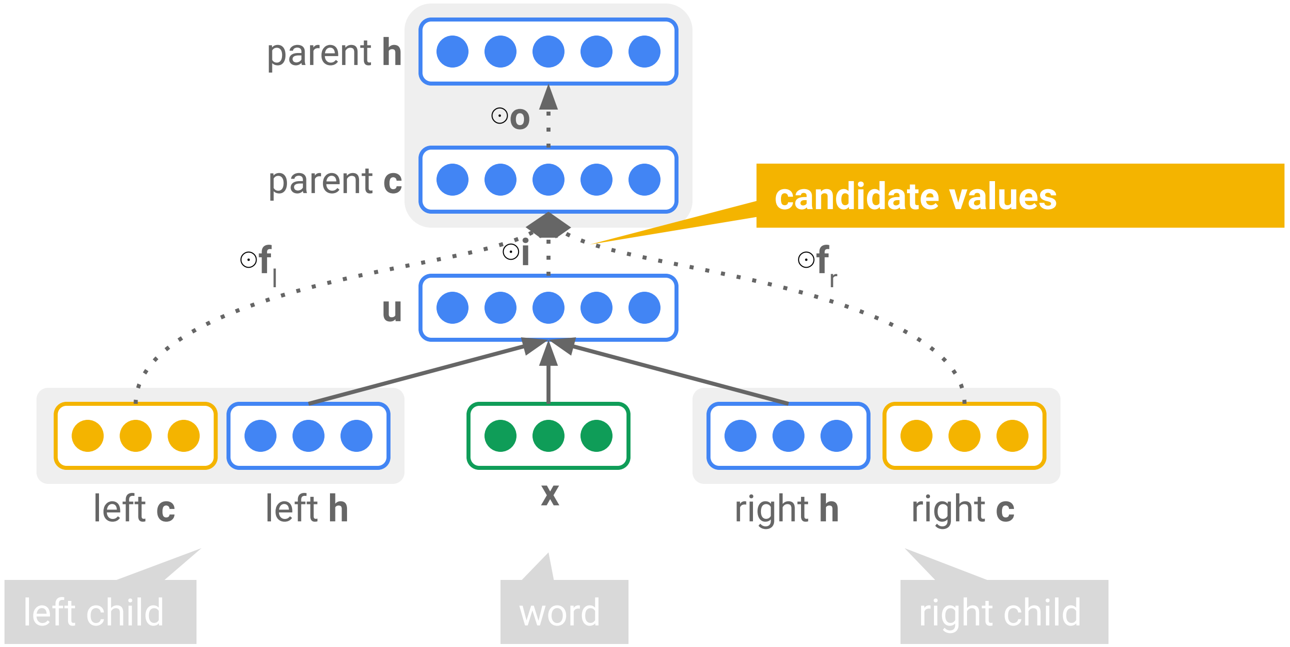
N-*ary* Tree LSTM

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

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

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

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

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

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

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

useful for encoding **constituency** trees

LSTMs vs Tree-LSTMs

Standard LSTMs be considered as (a special case of) Tree-LSTMs

Tree-LSTM variants

- **Child-Sum Tree-LSTM**
  - ➡ sum over the hidden representations of all children of a node (**no children order**)
  - ➡ can be used for a **variable** number of children
  - ➡ **shares parameters** between children
  - ➡ suitable for dependency trees

- **N-ary Tree-LSTM**
  - ➡ discriminates between **children node positions** (weighted sum)
  - ➡ **fixed** maximum branching factor: can be used with N children at most
  - ➡ **different parameters** for each child
  - ➡ suitable for constituency trees

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:

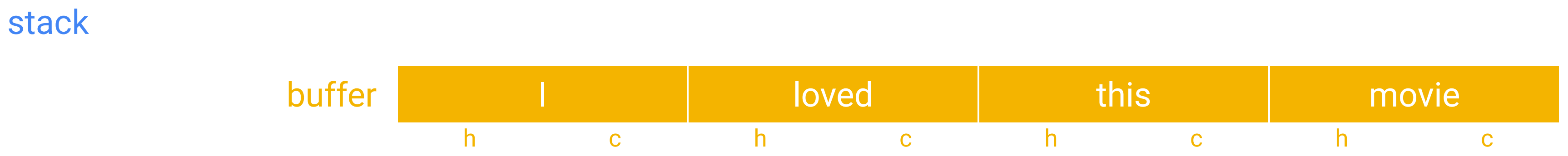
```
stack = []
buffer = queue([I, loved, this, movie])
```

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)**

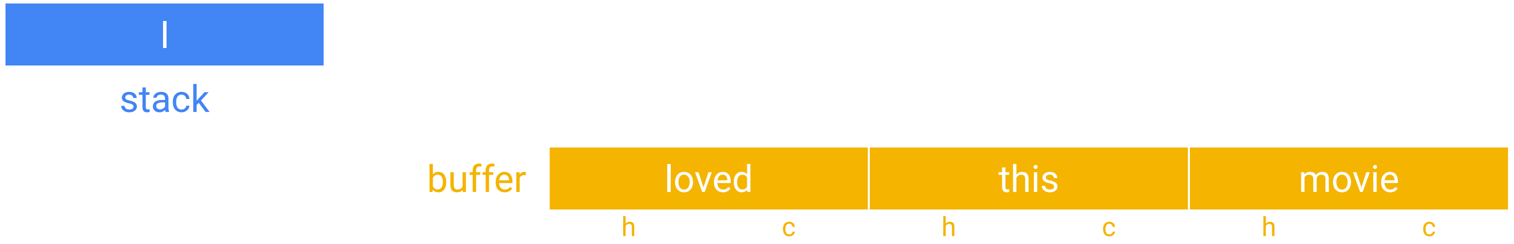
Transition sequence example

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



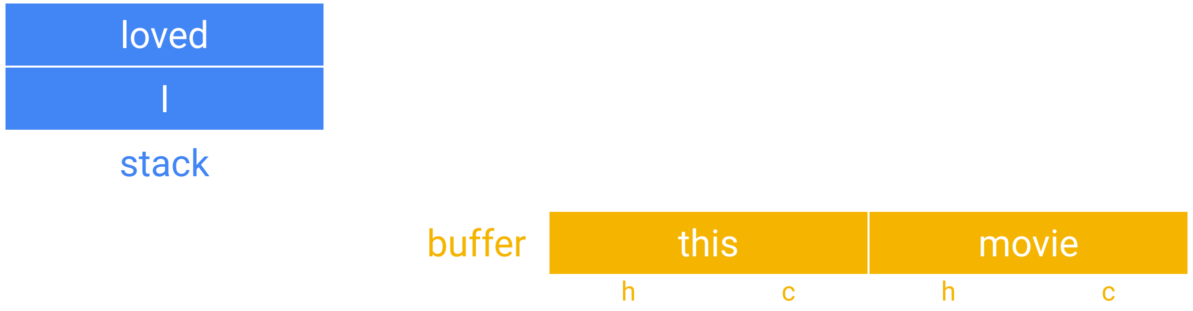
Transition sequence example

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



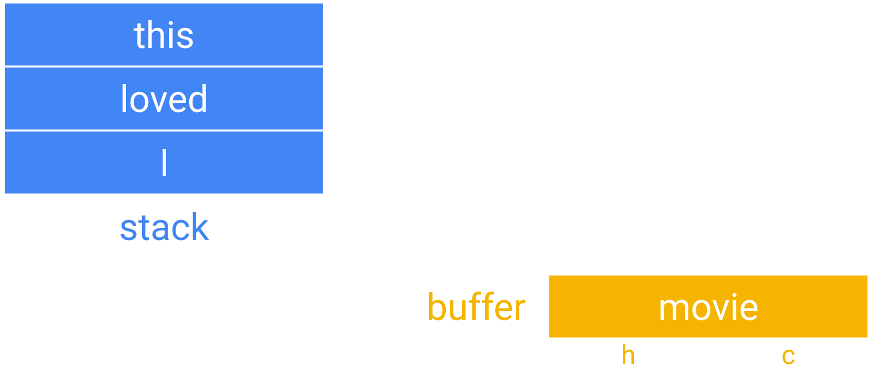
Transition sequence example

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



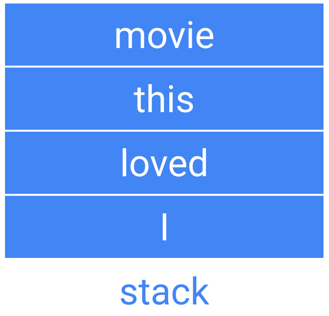
Transition sequence example

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



Transition sequence example

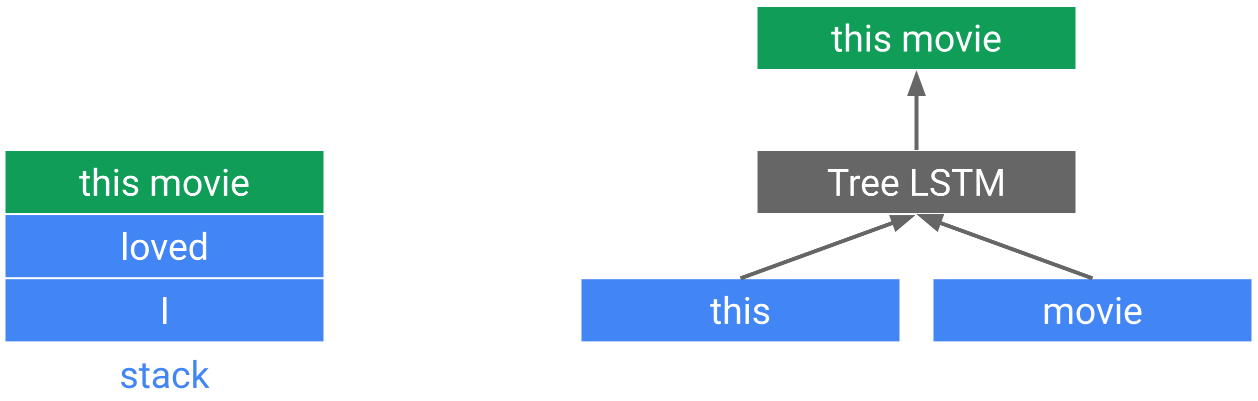
```
(I ( loved ( this movie ) ) )
 S   S         S    S       R  R  R
```



buffer

Transition sequence example

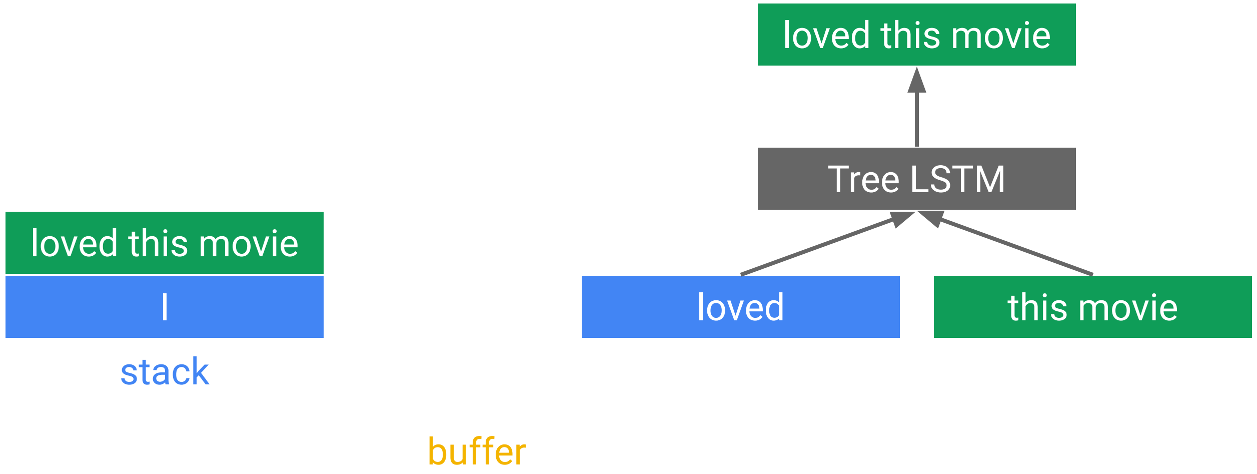
```
(I ( loved ( this movie ) ) )
 S   S        S    S      R R R
```



buffer

Transition sequence example

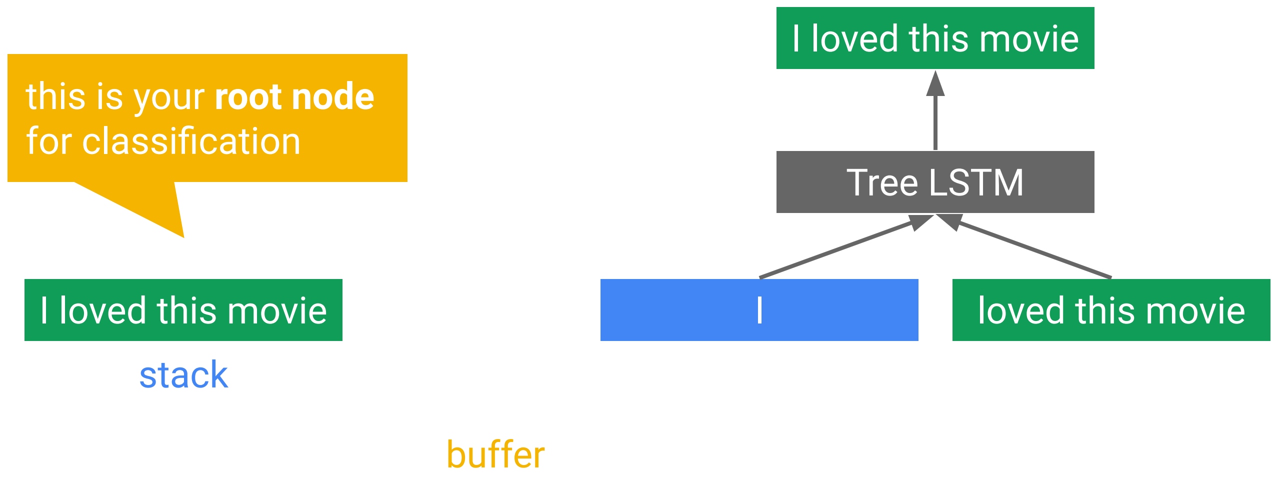
```
(I ( loved ( this movie ) ) )
 S   S       S    S       R R R
```



Transition sequence example

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

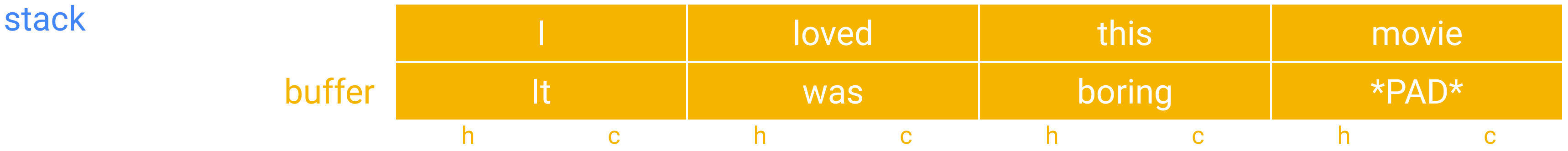
this is your **root node** for classification



Mini-batch SGD

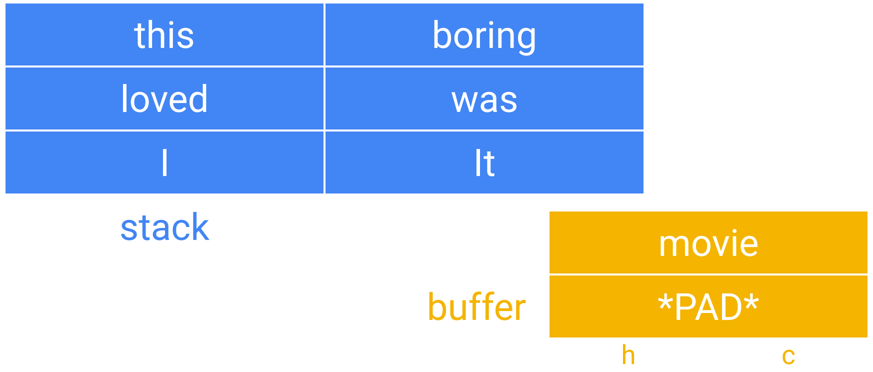
Transition sequence example (mini-batched)

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



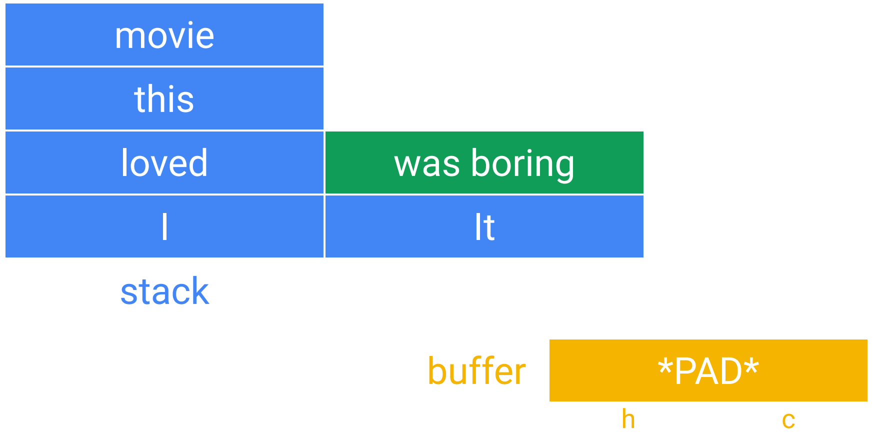
Transition sequence example (mini-batched)

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



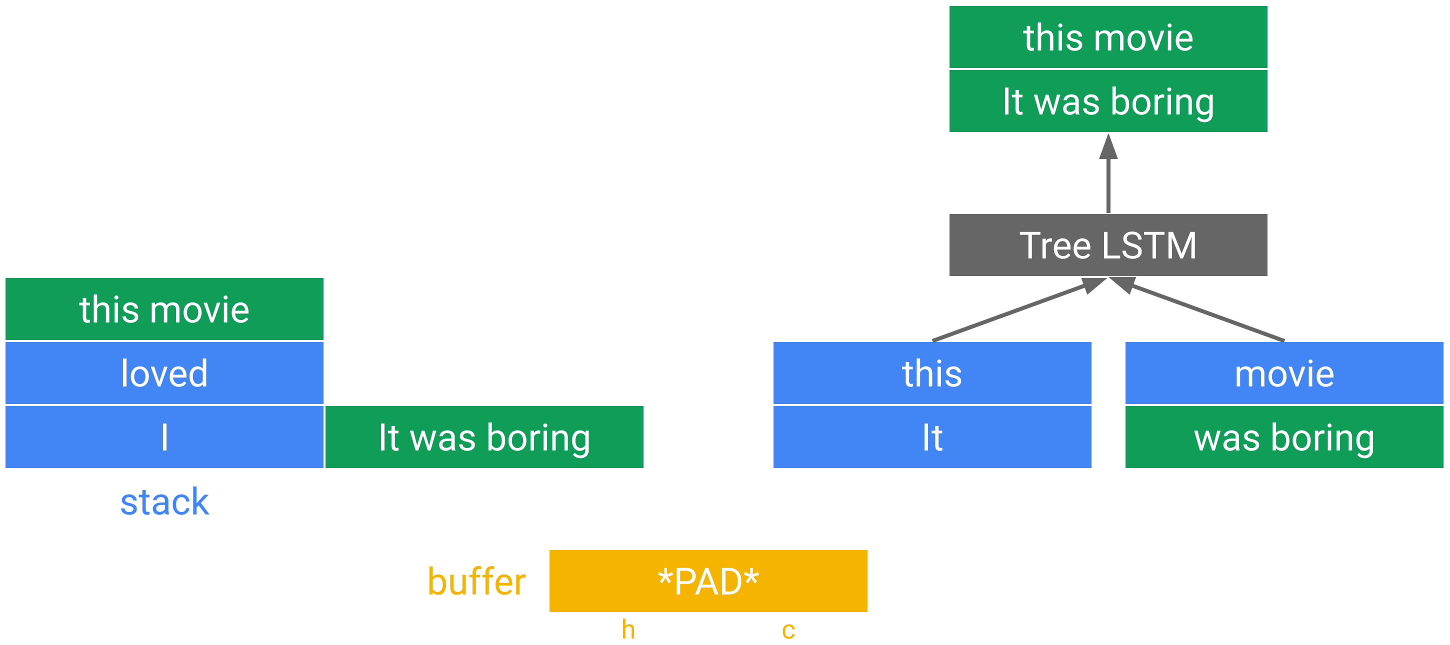
Transition sequence example (mini-batched)

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



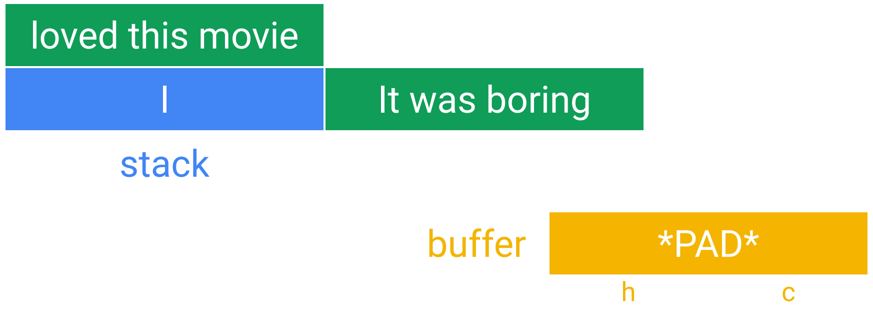
Transition sequence example (mini-batched)

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



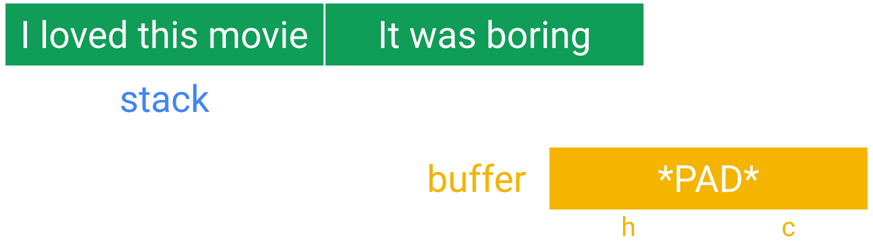
Transition sequence example (mini-batched)

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



Transition sequence example (mini-batched)

```
(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

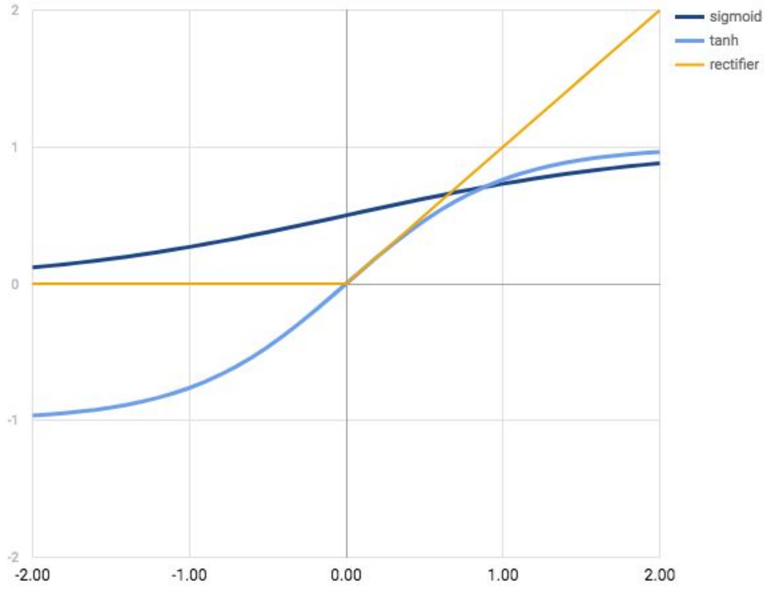
- Bag of Words models: BOW, CBOW, Deep CBOW
  - Can encode a sentence of arbitrary length, but loses word order
- Sequence models: RNN and LSTM
  - Sensitive to word order
  - RNN has vanishing gradient problem, LSTM deals with this
  - LSTM has input, forget, and output gates that control information flow
- Tree-based models: Child-Sum & N-ary Tree LSTM
  - Generalize LSTM to tree structures
  - Exploit compositionality, but require a parse tree

Extra

Input

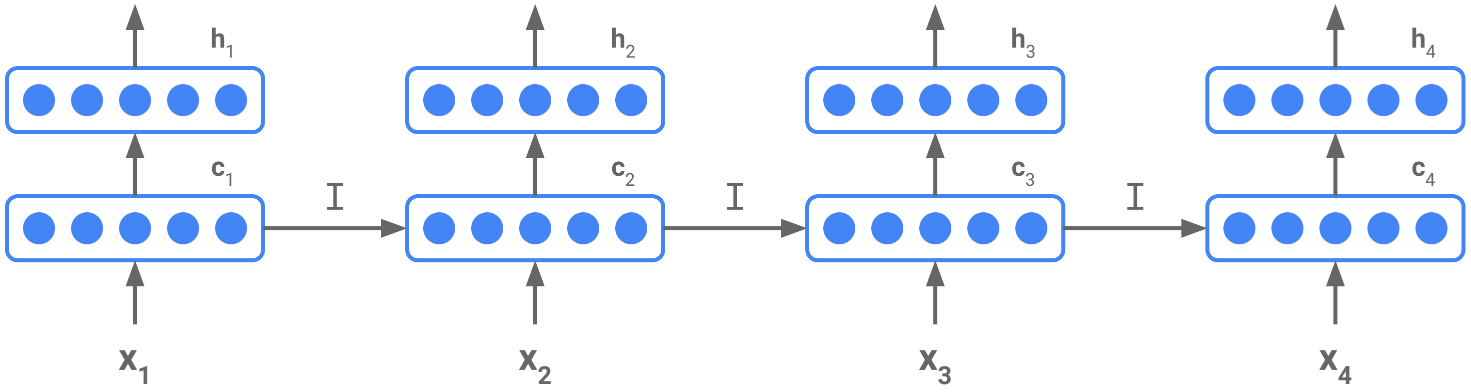
In a TreeLSTM over a constituency tree (ours!), the leaf nodes take the corresponding word vectors as input

Recap: Activation functions



Introduction: Intuition to solving the vanishing gradient

Let's use an extra vector, cell state **c**

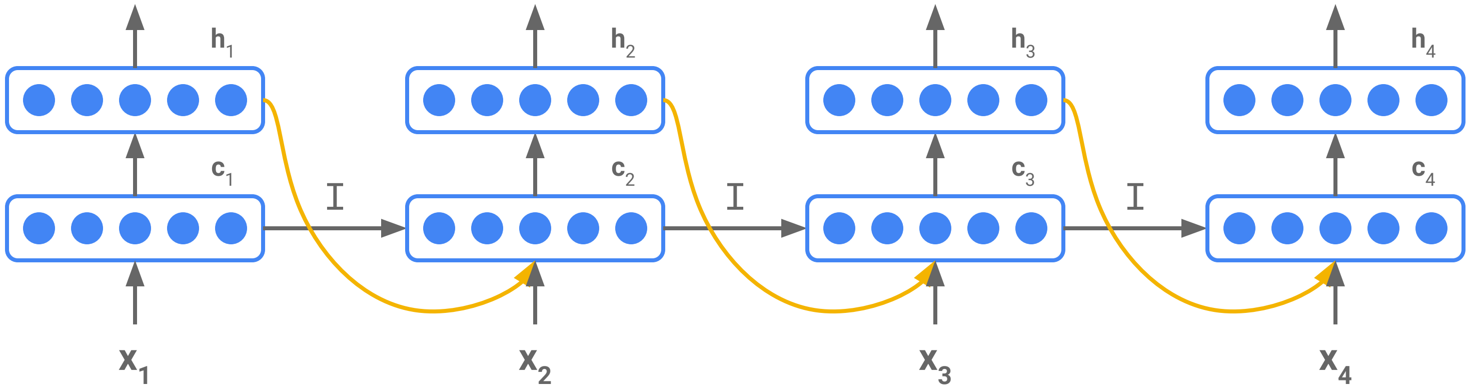


$\mathbf{c}_t = \mathbf{c}_{t-1} + f(\mathbf{x}_t)$ $\mathbf{h}_t = \tanh(\mathbf{c}_t)$ $$\frac{\delta \mathbf{c_t}}{\delta \mathbf{c_{t-1}}} = I$$

Adapted from Dyer, LxMLS 2016.

Introduction: A small improvement

Better gradient propagation is possible when you use **additive** rather than multiplicative/highly non-linear recurrent dynamics



$\mathbf{c}_t = \mathbf{c}_{t-1} + f(\mathbf{x}_{t,} \mathbf{h}_{t-1})$ $\mathbf{h}_t = \tanh(\mathbf{c}_t)$ $$\frac{\delta \mathbf{c_t}}{\delta \mathbf{c_{t-1}}} = I + \epsilon$$

Adapted from Dyer, LxMLS 2016.

Child-Sum Tree LSTM

useful for encoding **dependency** trees

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

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

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

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

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

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

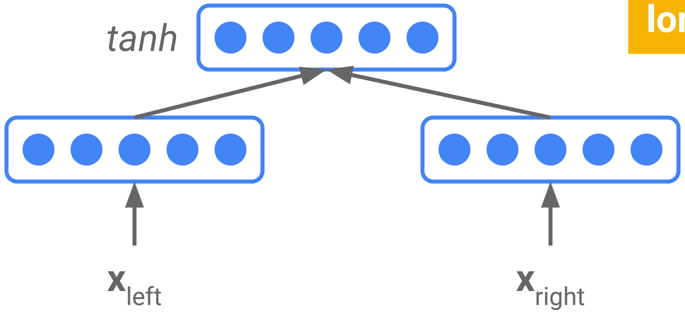
$$h_j = o_j \odot \tanh(c_j),$$

A naive recursive NN

Combine every two children (left and right) into a parent node **p:**

$$\mathbf{p} = \tanh(W_{left}\mathbf{x}_{left} + W_{right}\mathbf{x}_{right} + \mathbf{b})$$

a bit **simplistic** and does not work well for **longer sentences**



Richard Socher et al. Parsing natural scenes and natural language with recursive neural networks. ICML 2011.

SGD vs GD

Mini-batch SGD strikes a balance between these two

SGD:

```
for epoch in 1..E
  for each training example
    compute loss (forward pass)
    compute gradient of loss (backward)
    update parameters
  end for
end for
```

- **fast**, but **high variance**
- *might* find **better optimum** because of variance

Gradient Descent (GD):

```
for epoch in 1..E
  for each training example
    compute loss (forward pass)
    compute gradient of loss (backward)
    accumulate gradient
  end for
  update parameters
end for
```

- **slow**, but **more stable** (not overly influenced by most recent training example)
- **can get stuck in local optimum**

Source: Neubig.