Natural Language Processing 1

Lecture 7: Word embeddings and sentence representations

Katia Shutova

ILLC
University of Amsterdam

19 November 2018

Distributional semantic models

1. Count-based models:
    - Explicit vectors: dimensions are elements in the context
    - **long sparse** vectors with **interpretable** dimensions
2. Prediction-based models:
    - Train a model to predict plausible contexts for a word
    - learn word representations in the process
    - **short dense** vectors with **latent** dimensions

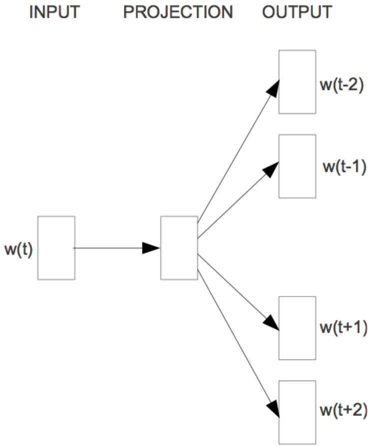
Prediction-based distributional models

Mikolov et. al. 2013. *Efficient Estimation of Word Representations in Vector Space*.

word2vec: **Skip-gram** model

- inspired by work on neural language models
- train a neural network to predict neighboring words
- learn dense embeddings for the words in the training corpus in the process

Skip-gram



Slide credit: Tomas Mikolov

Skip-gram

Intuition: words with similar meanings often occur near each other in texts

Given a word w_t:

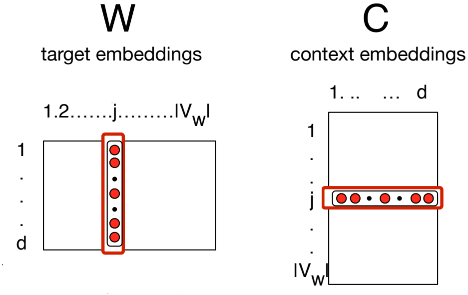
- Predict each neighbouring word
  - in a context window of $2L$ words
  - from the current word.
- For $L = 2$, we predict its 4 neighbouring words:

$$[w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}]$$

Skip-gram: Parameter matrices

Learn 2 embeddings for each word $w_j \in V_w$:

- **word embedding** v, in word matrix W
- **context embedding** c, in context matrix C



Skip-gram: Setup

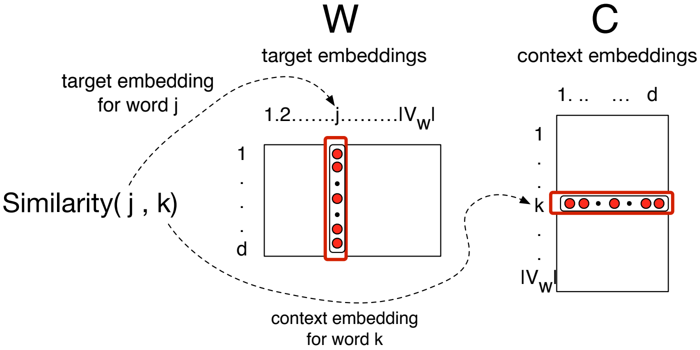
- Walk through the corpus pointing at word $w(t)$, whose index in the vocabulary is j — we will call it w_j
- our goal is to predict $w(t+1)$, whose index in the vocabulary is k — we will call it w_k
- to do this, we need to compute

$$p(w_k|w_j)$$

- Intuition behind skip-gram: to compute this probability we need to compute similarity between w_j and w_k

Skip-gram: Computing similarity

Similarity as dot-product between the target vector and context vector



Slide credit: Dan Jurafsky

Skip-gram: Similarity as dot product

- Remember cosine similarity?

$$cos(v1, v2) = \frac{\sum v1_k * v2_k}{\sqrt{\sum v1_k^2} * \sqrt{\sum v2_k^2}} = \frac{v1 \cdot v2}{||v1||||v2||}$$

It's just a normalised dot product.

- Skip-gram: Similar vectors have a high dot product

$$Similarity(c_k, v_j) \propto c_k \cdot v_j$$

Skip-gram: Compute probabilities

- Compute similarity as a dot product

$$Similarity(c_k, v_j) \propto c_k \cdot v_j$$

- Normalise to turn this into a probability
- by passing through a softmax function:

$$p(w_k|w_j) = \frac{e^{c_k \cdot v_j}}{\sum_{i \in V} e^{c_i \cdot v_j}}$$

Skip-gram: Learning

- Start with some initial embeddings (usually random)
- At training time, walk through the corpus
- iteratively make the embeddings for each word
  - more like the embeddings of its neighbors
  - less like the embeddings of other words.

Skip-gram: Objective

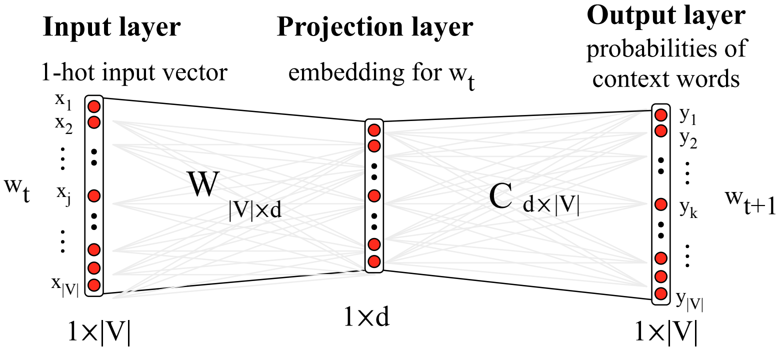
Learn parameters C and W that maximize the overall corpus probability:

$$\arg\max \prod_{(w_j,w_k)\in D} p(w_k|w_j)$$

$$p(w_k|w_j) = \frac{e^{c_k \cdot v_j}}{\sum_{i\in V} e^{c_i \cdot v_j}}$$

$$\arg\max \sum_{(w_j,w_k)\in D} \log p(w_k|w_j) = \sum_{(w_j,w_k)\in D} (\log e^{c_k \cdot v_j} - \log \sum_{c_i\in V} e^{c_i \cdot v_j})$$

Visualising skip-gram as a network



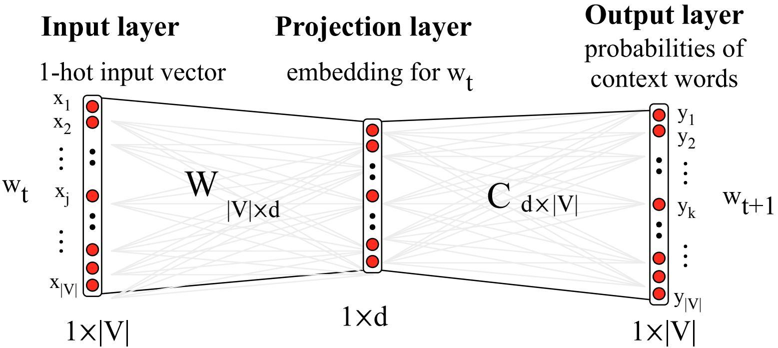
Slide credit: Dan Jurafsky

One hot vectors

- A vector of length |V|
- 1 for the target word and 0 for other words
- So if “bear” is vocabulary word 5
- The one-hot vector is [0,0,0,0,1,0,0,0,0........0]

$$\begin{array}{ccccccccccccccccccccc} w_0 & w_1 & & & & & & & & & w_j & & & & & & & & & & w_{|V|} \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \end{array}$$

Visualising skip-gram as a network



Slide credit: Dan Jurafsky

Skip-gram with negative sampling

Problem with softmax: expensive to compute the denominator for the whole vocabulary

$$p(w_k|w_j) = \frac{e^{c_k \cdot v_j}}{\sum_{i \in V} e^{c_i \cdot v_j}}$$

Approximate the denominator: **negative sampling**

- At training time, walk through the corpus
- for each target word and positive context
- sample *k* noise samples or negative samples, i.e. other words

Skip-gram with negative sampling

- Objective in training:
  - Make the word like the context words
    lemon, a [tablespoon of apricot preserves or] jam.

    c_1 c_2 w c_3 c_4
  - And not like the k negative examples

[cement idle dear coaxial apricot attendant whence forever puddle]

n_1 n_2 n_3 n_4 w n_5 n_6 n_7 n_8

Skip-gram with negative sampling: Training examples

Convert the dataset into word pairs:

- **Positive (+)**

  (apricot, tablespoon)
  (apricot, of)
  (apricot, jam)
  (apricot, or)

- **Negative (-)**

  (apricot, cement)
  (apricot, idle)
  (apricot, attendant)
  (apricot, dear)
  ...

Skip-gram with negative sampling

- instead of treating it as a **multi-class problem** (and returning a probability distribution over the whole vocabulary)
- **return a probability** that word w_k is a valid context for word w_j

$$P(+|w_j, w_k)$$

$$P(-|w_j, w_k) = 1 - P(+|w_j, w_k)$$

Skip-gram with negative sampling

- model similarity as dot product

$$Similarity(c_k, v_j) \propto c_k \cdot v_j$$

- turn this into a probability using the **sigmoid function**:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$P(+|w_j, w_k) = \frac{1}{1 + e^{-c_k \cdot v_j}}$$

$$P(-|w_j, w_k) = 1 - P(+|w_j, w_k) = 1 - \frac{1}{1 + e^{-c_k \cdot v_j}} = \frac{1}{1 + e^{c_k \cdot v_j}}$$

Skip-gram with negative sampling

- model similarity as dot product

$$Similarity(c_k, v_j) \propto c_k \cdot v_j$$

- turn this into a probability using the **sigmoid function**:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$P(+|w_j, w_k) = \frac{1}{1 + e^{-c_k \cdot v_j}}$$

$$P(-|w_j, w_k) = 1 - P(+|w_j, w_k) = 1 - \frac{1}{1 + e^{-c_k \cdot v_j}} = \frac{1}{1 + e^{c_k \cdot v_j}}$$

Skip-gram with negative sampling: Objective

- make the word like the context words
- and not like the negative examples

$$\arg\max \prod_{(w_j,w_k)\in D_+} p(+|w_k, w_j) \prod_{(w_j,w_k)\in D_-} p(-|w_k, w_j)$$

$$\arg\max \sum_{(w_j,w_k)\in D_+} \log p(+|w_k, w_j) + \sum_{(w_j,w_k)\in D_-} \log p(-|w_k, w_j) =$$

$$\arg\max \sum_{(w_j,w_k)\in D_+} \log \frac{1}{1+e^{-c_k \cdot v_j}} + \sum_{(w_j,w_k)\in D_-} \log \frac{1}{1+e^{c_k \cdot v_j}}$$

Skip-gram with negative sampling: Objective

- make the word like the context words
- and not like the negative examples

$$\arg\max \prod_{(w_j,w_k)\in D_+} p(+|w_k, w_j) \prod_{(w_j,w_k)\in D_-} p(-|w_k, w_j)$$

$$\arg\max \sum_{(w_j,w_k)\in D_+} \log p(+|w_k, w_j) + \sum_{(w_j,w_k)\in D_-} \log p(-|w_k, w_j) =$$

$$\arg\max \sum_{(w_j,w_k)\in D_+} \log \frac{1}{1+e^{-c_k\cdot v_j}} + \sum_{(w_j,w_k)\in D_-} \log \frac{1}{1+e^{c_k\cdot v_j}}$$

Skip-gram with negative sampling: Objective

- make the word like the context words
- and not like the negative examples

$$\arg\max \prod_{(w_j,w_k)\in D_+} p(+|w_k, w_j) \prod_{(w_j,w_k)\in D_-} p(-|w_k, w_j)$$

$$\arg\max \sum_{(w_j,w_k)\in D_+} \log p(+|w_k, w_j) + \sum_{(w_j,w_k)\in D_-} \log p(-|w_k, w_j) =$$

$$\arg\max \sum_{(w_j,w_k)\in D_+} \log \frac{1}{1+e^{-c_k \cdot v_j}} + \sum_{(w_j,w_k)\in D_-} \log \frac{1}{1+e^{c_k \cdot v_j}}$$

Properties of embeddings

They capture similarity

FRANCE 454	JESUS 1973	XBOX 6909	REDDISH 11724	SCRATCHED 29869	MEGABITS 87025
AUSTRIA	GOD	AMIGA	GREENISH	NAILED	OCTETS
BELGIUM	SATI	PLAYSTATION	BLUISH	SMASHED	MB/S
GERMANY	CHRIST	MSX	PINKISH	PUNCHED	BIT/S
ITALY	SATAN	IPOD	PURPLISH	POPPED	BAUD
GREECE	KALI	SEGA	BROWNISH	CRIMPED	CARATS
SWEDEN	INDRA	PSNUMBER	GREYISH	SCRAPED	KBIT/S
NORWAY	VISHNU	HD	GRAYISH	SCREWED	MEGAHERTZ
EUROPE	ANANDA	DREAMCAST	WHITISH	SECTIONED	MEGAPIXELS
HUNGARY	PARVATI	GEFORCE	SILVERY	SLASHED	GBIT/S
SWITZERLAND	GRACE	CAPCOM	YELLOWISH	RIPPED	AMPERES

Slide credit: Ronan Collobert

Properties of embeddings

They capture analogy

Analogy task: ***a*** *is to* ***b*** *as* ***c*** *is to* ***d***
The system is given words a, b, c, and it needs to find d.

“apple” is to “apples” as “car”‘ is to ?
“man” is to “woman” as “king” is to ?

Solution: capture analogy via vector offsets

$$a - b \approx c - d$$

$$man - woman \approx king - queen$$

$$d_w = \operatorname*{argmax}_{d'_w \in V} cos(a - b, c - d')$$

Properties of embeddings

They capture analogy

Analogy task: ***a*** *is to* ***b*** *as* ***c*** *is to* ***d***
The system is given words a, b, c, and it needs to find d.

"apple" is to "apples" as "car"" is to ?
"man" is to "woman" as "king" is to ?

Solution: capture analogy via vector offsets

$$a - b \approx c - d$$

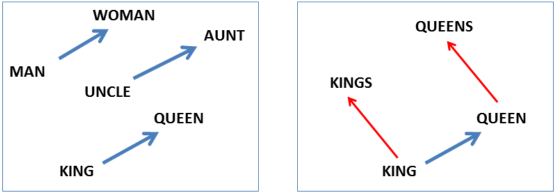
$$man - woman \approx king - queen$$

$$d_w = \underset{d'_w \in V}{\operatorname{argmax}} \, cos(a - b, c - d')$$

Properties of embeddings

Capture analogy via vector offsets

$$\textit{man} - \textit{woman} \approx \textit{king} - \textit{queen}$$



Mikolov et al. 2013. *Linguistic Regularities in Continuous Space Word Representations*

Properties of embeddings

They capture a range of semantic relations

Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

Mikolov et al. 2013. *Efficient Estimation of Word Representations in Vector Space*

Word embeddings in practice

Word2vec is often used for pretraining in other tasks.

- It will help your models start from an **informed** position
- Requires only **plain text** - which we have a lot of
- Is very **fast** and easy to use
- Already **pretrained** vectors also available (trained on 100B words)

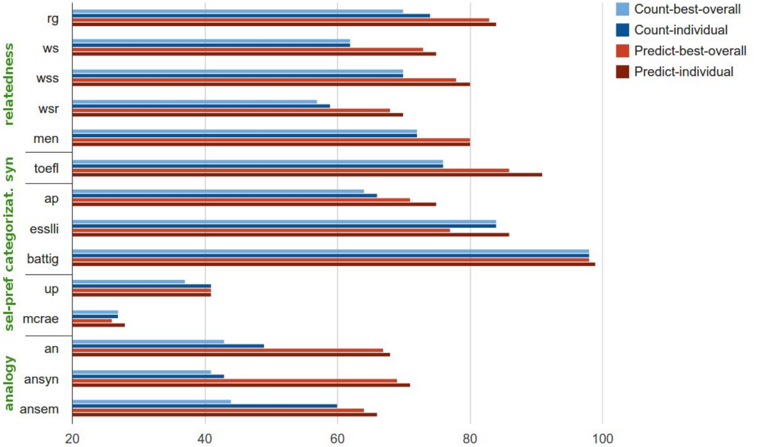
However, for best performance it is important to continue training, fine-tuning the embeddings for a specific task.

Count-based models vs. skip-gram word embeddings

Baroni et. al. 2014. *Don't count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors.*

- Comparison of count-based and neural word vectors on 5 types of tasks and 14 different datasets:
    1. Semantic relatedness
    2. Synonym detection
    3. Concept categorization
    4. Selectional preferences
    5. Analogy recovery

Count-based models vs. skip-gram word embeddings



Some of these findings were later disputed by Levy et. al. 2015. *Improving Distributional Similarity with Lessons Learned from Word Embeddings*

Acknowledgement

Some slides were adapted from Dan Jurafsky

UNIVERSITY OF AMSTERDAM

Encoding Sentences with **Recurrent** and **Tree Recursive** Neural Networks

Joost Bastings

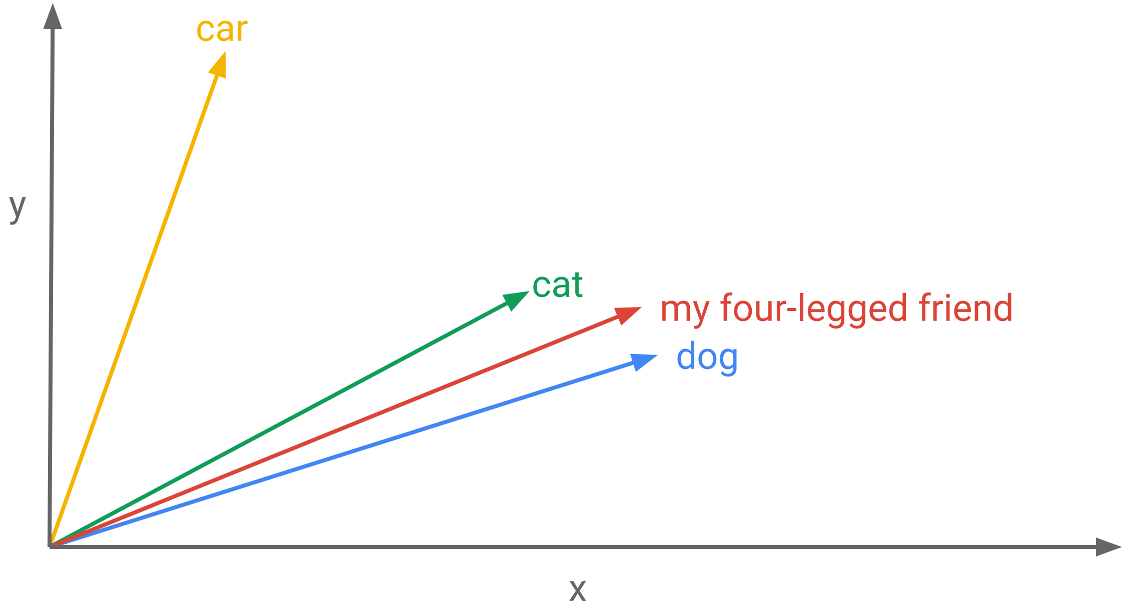
bastings.github.io

Today

How do we learn a **representation** of a **sentence** with a **neural network**?

How do we make a **prediction** from that representation, e.g. **sentiment**?

A vector space of words and sentences



Turning words into numbers

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

Bad idea: number sequence

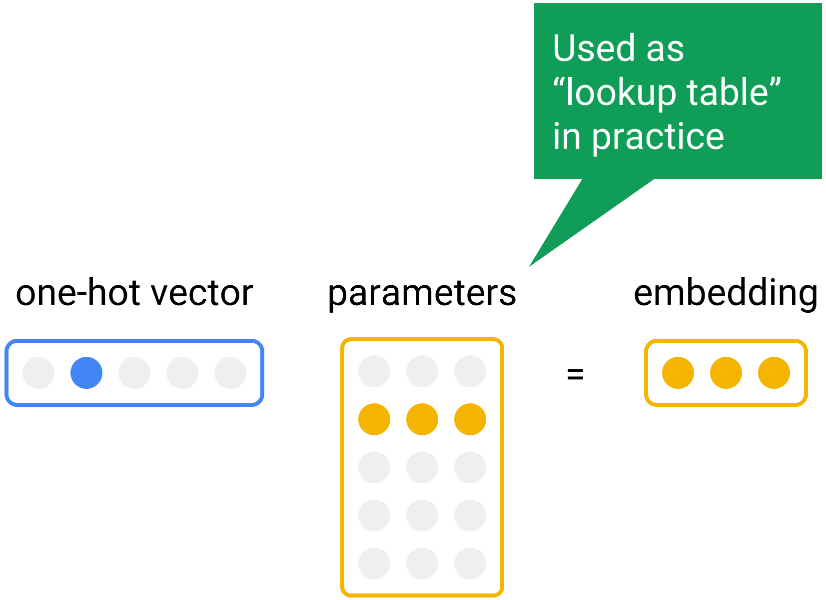
`cat`	`1`
`tree`	`2`
`chair`	`3`
`dog`	`4`
`mat`	`5`

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

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]`

One-hot vectors select word embeddings



Used as "lookup table" in practice

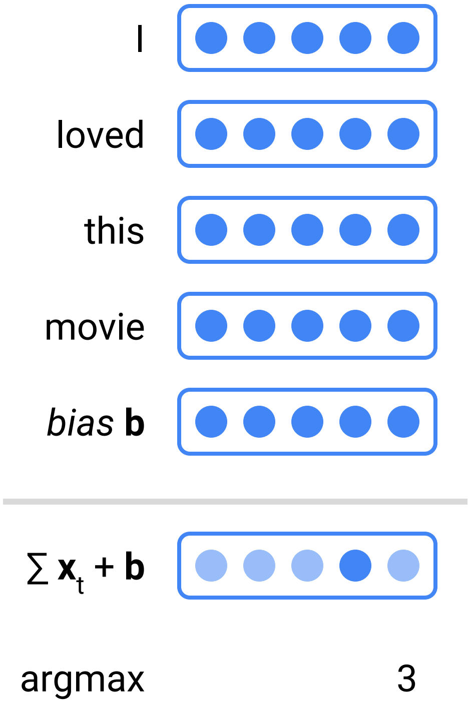
Bag of Words

Bag of Words at CMU

https://tinyurl.com/cmu-bagofwords

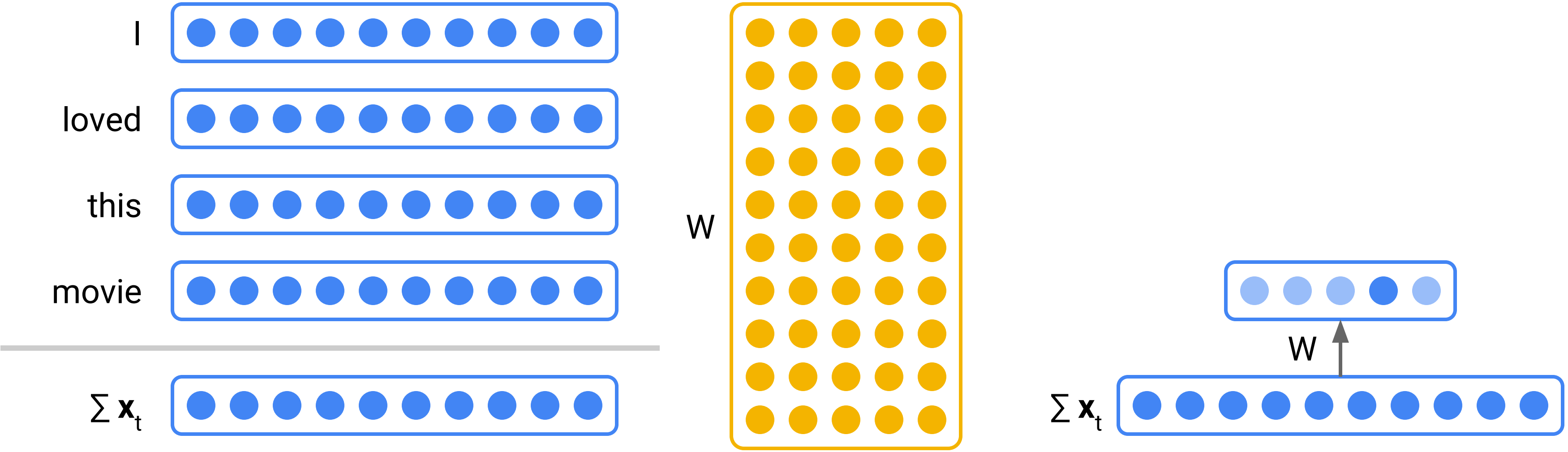
Bag of Words

Sum word embeddings, add bias



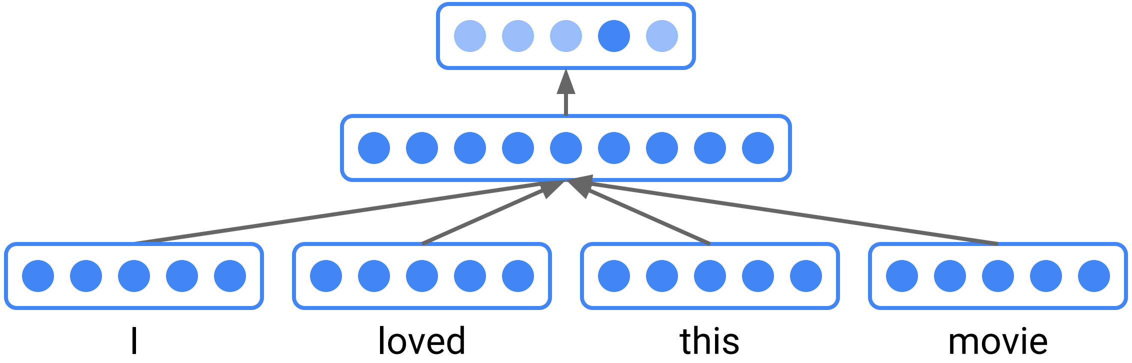
Continuous Bag of Words (CBOW)

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



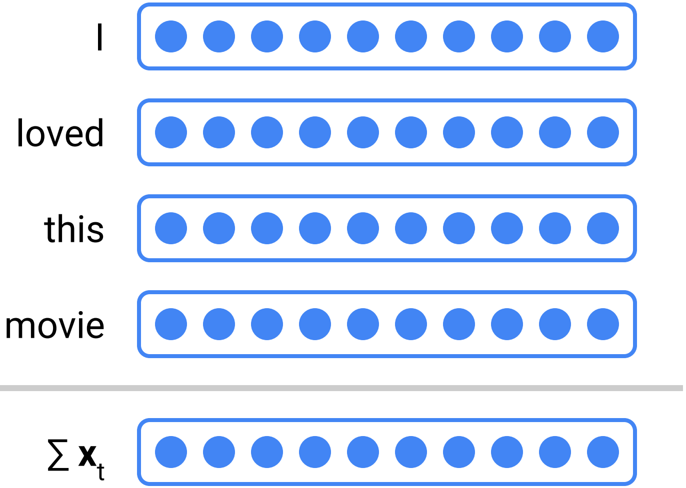
Note that a bias term (of size 5) is added to the final output vector (not shown). Also, this is not the same as word2vec CBOW!

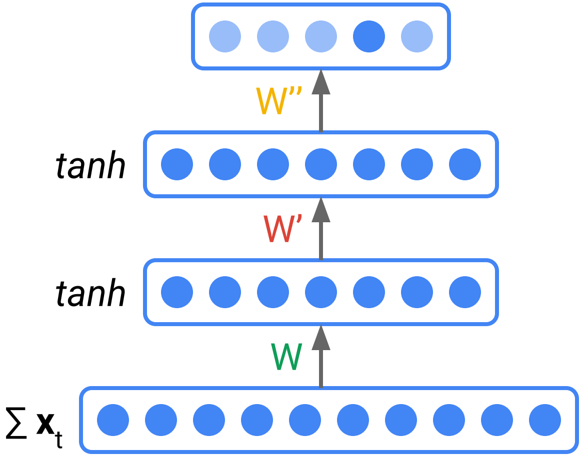
Why not this?



Deep CBOW

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





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

Softmax

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

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

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

This makes **o** sum to 1.0:

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

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**
4. Backward pass (backpropagation)
    a. Compute gradient of loss w.r.t. parameters
5. Take a small step in the opposite direction of the gradient

Cross Entropy Loss

Given:

$\hat{\mathbf{y}}$ = `[0.1, 0.1, 0.1, 0.5, 0.2]` *output vector (after softmax) from forward pass*

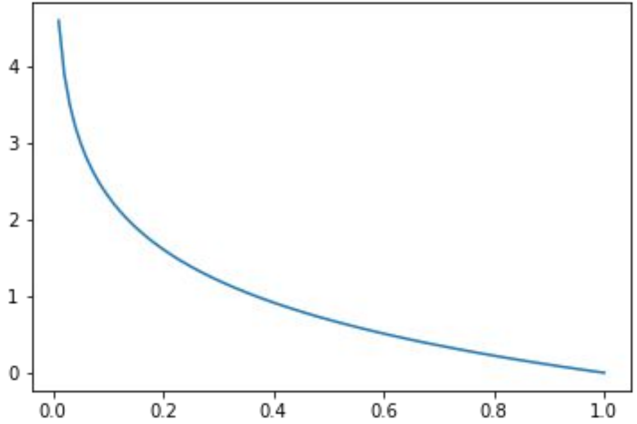
$\mathbf{y}$ = `[  0,   0,   0,   1,   0]` *target / label ($y_3 = 1$)*

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!

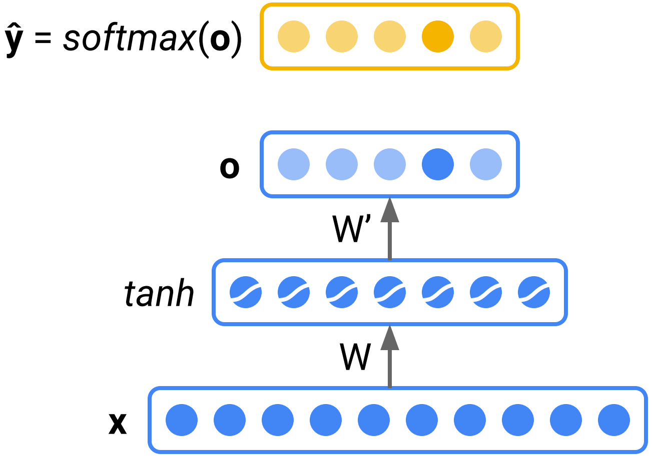


Backpropagation example

the **chain rule** is your friend!

$L = f(g(x))$

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



$\hat{\mathbf{y}} = [0.1, 0.1, 0.1, 0.5, 0.2]$

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

loss $L = CE(\hat{\mathbf{y}}, \mathbf{y}) = -\log(\hat{y}_3) = -\log(0.5)$

compute gradients, e.g. for W':

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

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

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

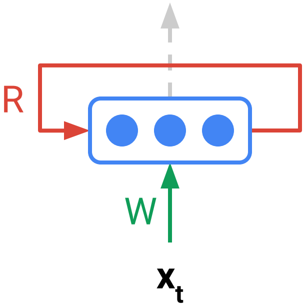
update weights:

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

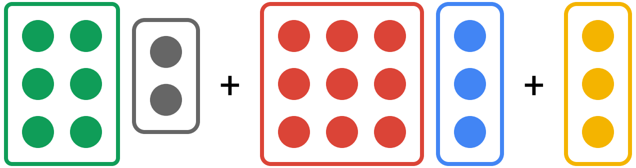
Sequences

Recurrent Neural Network (RNN)

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



$$\begin{aligned}\mathbf{h}_t &= f(\mathbf{x}_t, \mathbf{h}_{t-1}) \\ &= \sigma(W\mathbf{x}_t + R\mathbf{h}_{t-1} + \mathbf{b})\end{aligned}$$



Elman (1990). Finding structure in time.

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:

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

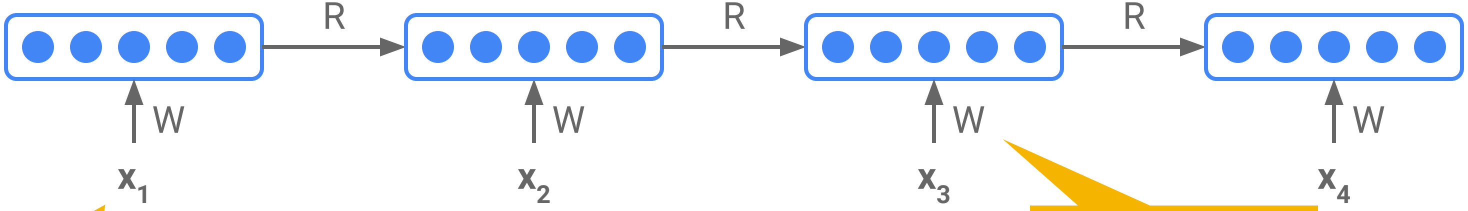
$$\mathbf{h}_1 = f(\mathbf{x}_1, \mathbf{h}_0)$$
$$\mathbf{h}_2 = f(\mathbf{x}_2, f(\mathbf{x}_1, \mathbf{h}_0))$$
$$\mathbf{h}_3 = f(\mathbf{x}_3, f(\mathbf{x}_2, f(\mathbf{x}_1, \mathbf{h}_0)))$$

…

$$\mathbf{h}_6 = f(\mathbf{x}_6, f(\mathbf{x}_5, f(\mathbf{x}_4, \ldots)))$$

Elman (1990)

Unfolding the RNN

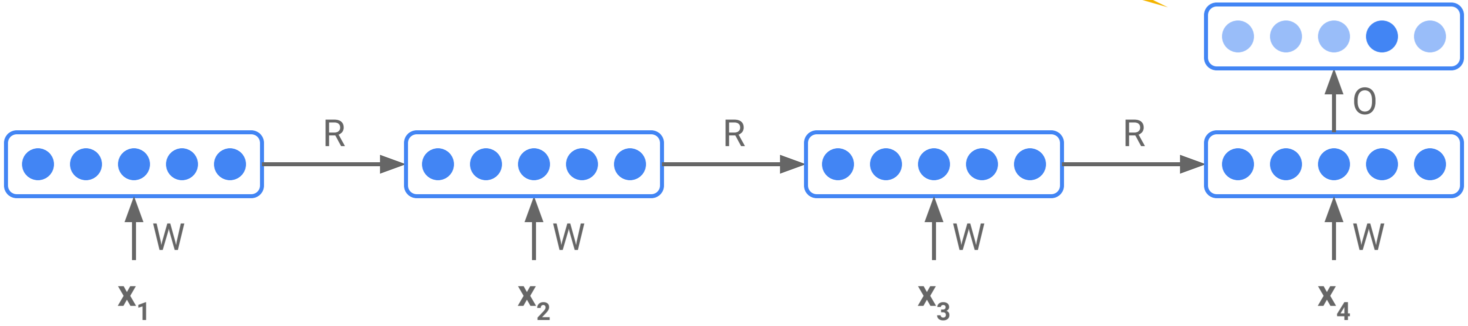


Same R every time step!

Same W every time step!

Word embedding

Making a prediction



We can find the **prediction** using **argmax**

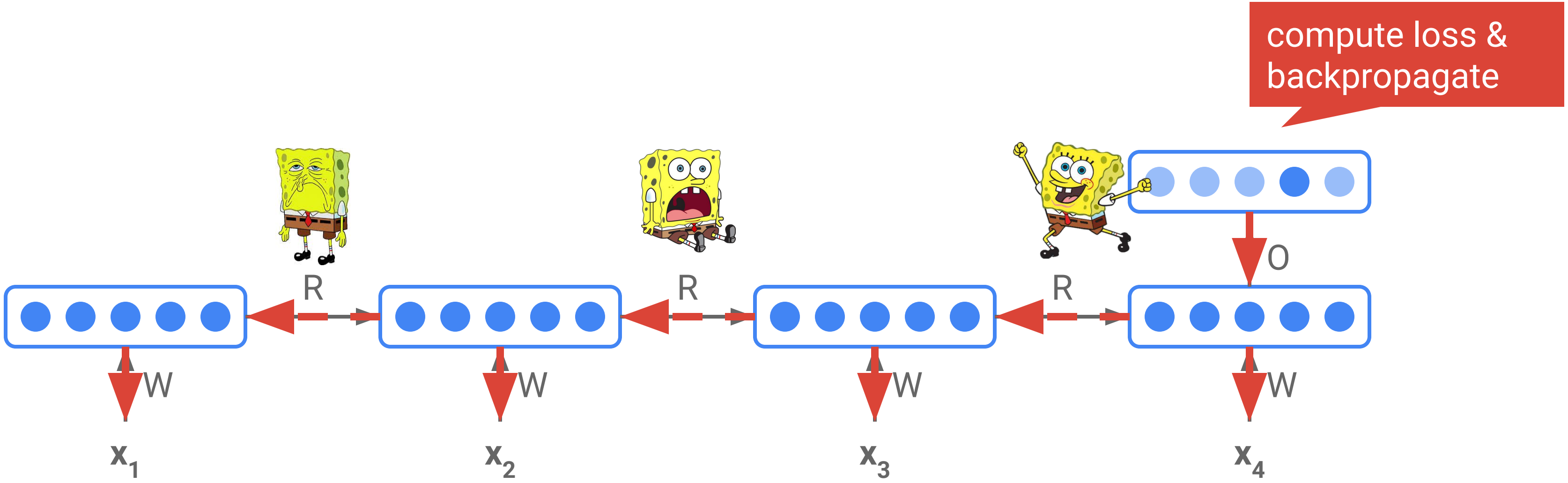
Training:
apply **softmax,**
compute **cross entropy** loss,
backpropagate

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)

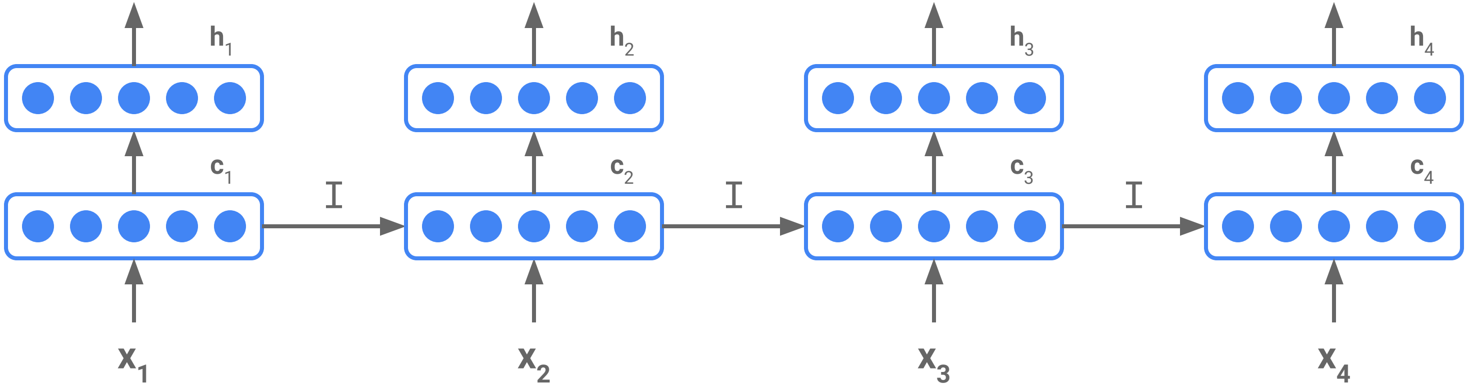


compute loss & backpropagate

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

Intuition to solving the vanishing gradient

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



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

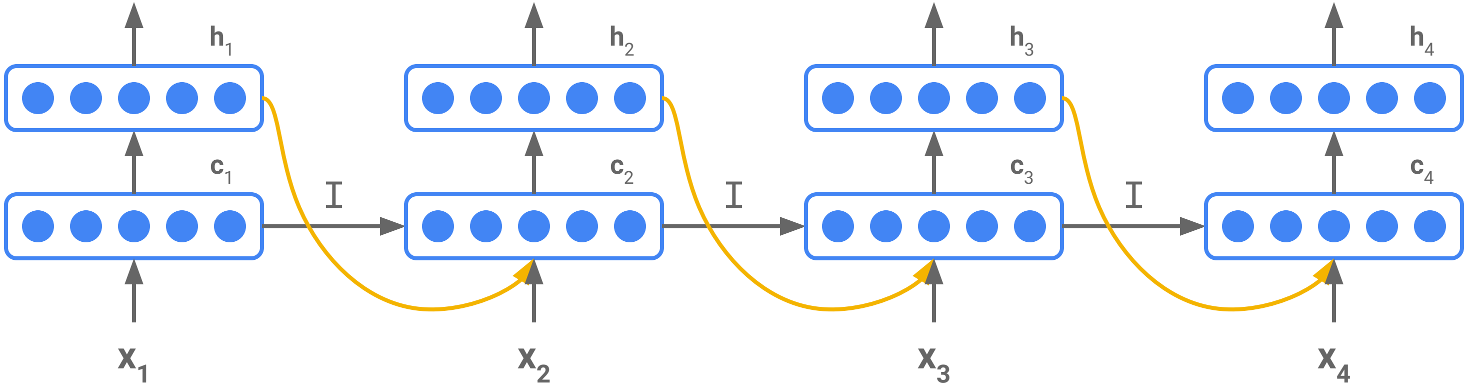
$\mathbf{h}_t = \tanh(\mathbf{c}_t)$

$$\frac{\delta \mathbf{c_t}}{\delta \mathbf{c_{t-1}}} = I$$

Adapted from Dyer, LxMLS 2016.

A small improvement

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



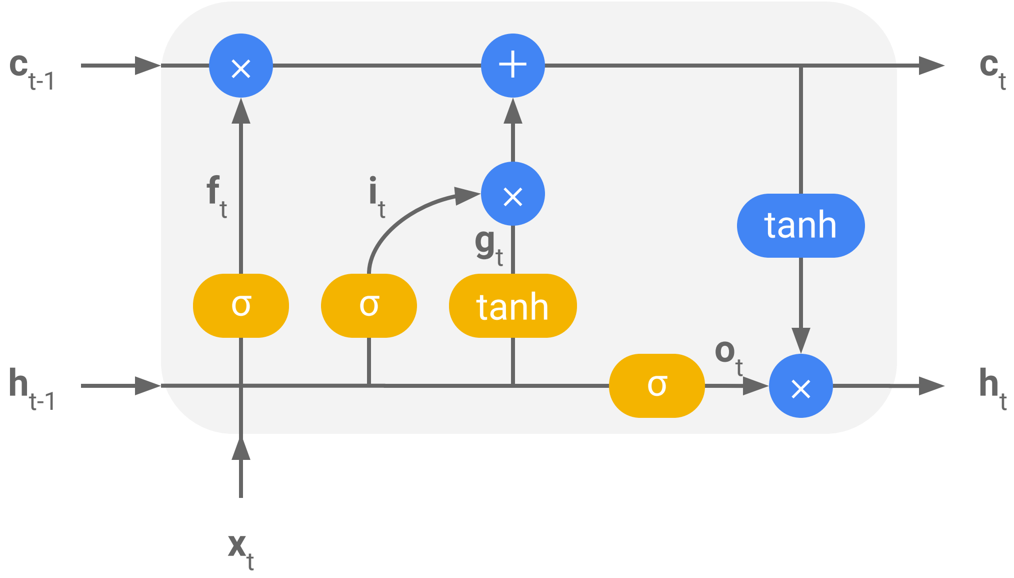
$\mathbf{c}_t = \mathbf{c}_{t-1} + f(\mathbf{x}_t, \mathbf{h}_{t-1})$ $\mathbf{h}_t = \tanh(\mathbf{c}_t)$

$$\frac{\delta \mathbf{c_t}}{\delta \mathbf{c_{t-1}}} = I + \epsilon$$

Adapted from Dyer, LxMLS 2016.

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)

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

Trees

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**.

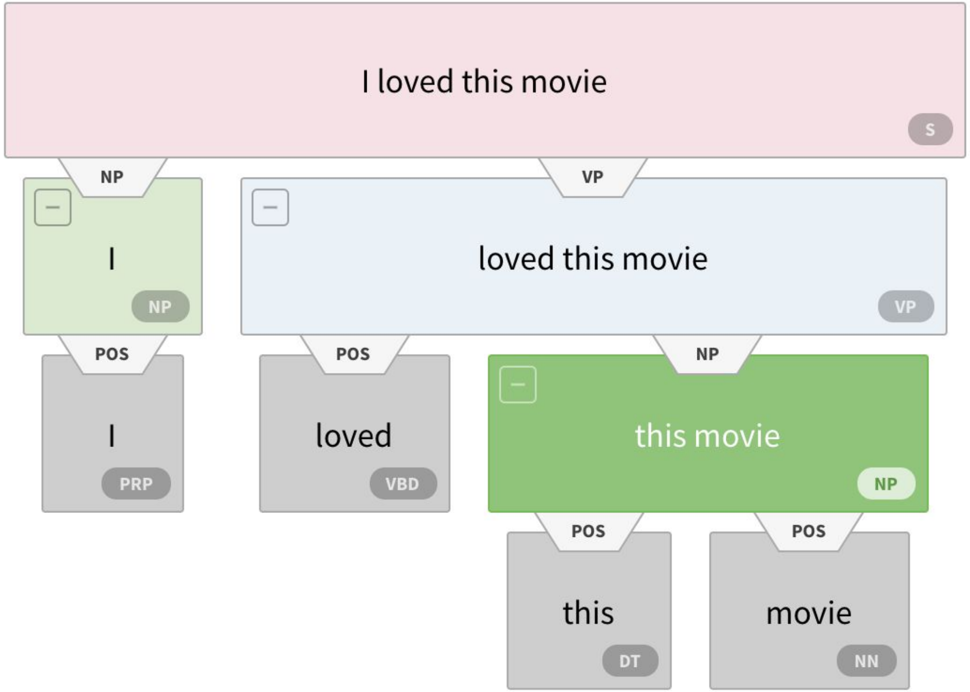
The meaning (vector) of a sentence is determined by:

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

Adapted from Stanford cs224n.

Constituency Parse

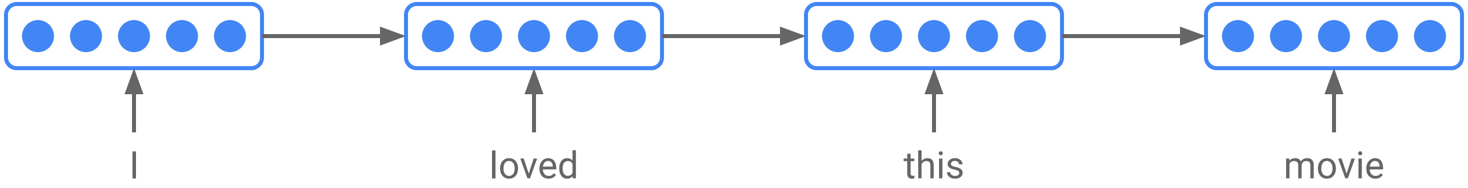
Can we obtain a sentence vector using the tree structure given by a parse?



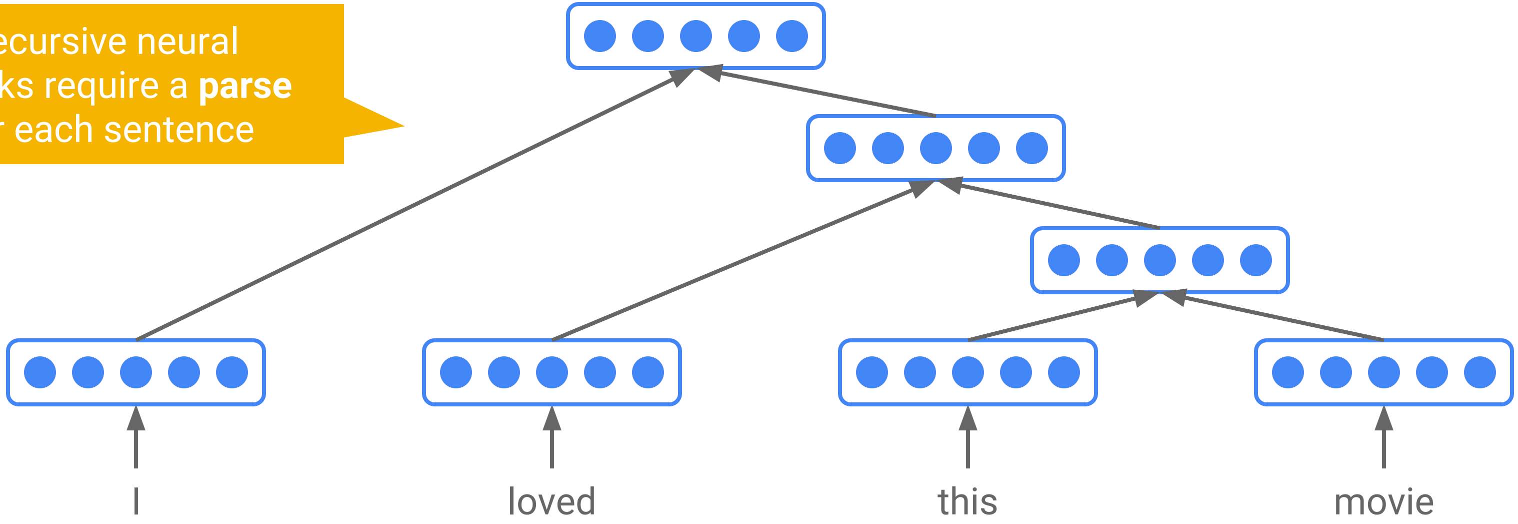
http://demo.allennlp.org/constituency-parsing

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



Adapted from Stanford cs224n.

Practical II data set: Stanford Sentiment Treebank (SST)

sentiment label for **root node**

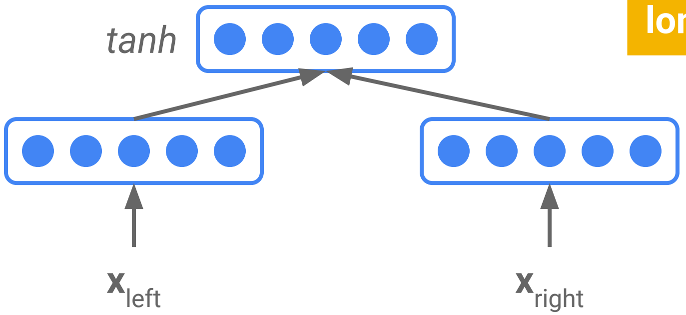
```
             3
 ------------|--------------------
|                                 4
|        -------------------------|----------------------------------------------
|       4                                                                        |
|    ---|--------------                                                          |
|   |                  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  .
```

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.

Tree LSTM next time!

Extra

Recap: Matrix Multiplication

Rows multiply with **columns**

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

Recap: Activation functions