Faculty for Informatics

Technical University of Munich

Natural Language Processing
IN2361

PD Dr. Georg Groh

Social Computing
Research Group

Deep NLP
Part A: Introduction and Short Repetition of DL Basics

- content is based on [1] and [2](lectures 1-9) and [10](lectures 1-8) (further sources: see bibliography)
  [1] is the primary source; slides mostly based on / or taken over from [2] and [10] are marked with ● to allow students to refer the original source more easily when working with the slides
- certain elements (e.g. figures, equations or tables) were taken over or taken over in a modified form from [1] or [2] or [10]
- citations of [1] and [2] and [10] are omitted for legibility; non-[1]or[2]or[10]-sources are cited
- citations of original sources cited in [1] and [2] and [10] are omitted for legibility
- errors on these slides are fully in the responsibility of Georg Groh
- BIG thanks to Yoav Goldberg for a great review / primer article [1]!
- BIG thanks to Richard Socher and his colleagues at Stanford for publishing materials [2] of a great Deep NLP lecture
- BIG thanks to Christopher Manning and his colleagues at Stanford for publishing materials [10] of a great Deep NLP lecture

Introduction

- What we have seen so far: very many NLP problems may be approached with ML (e.g. sequence classifiers)
- Many "traditional" ML models (MEMMs, HMMs, linear classifiers (linear Regression, SVMs) etc.) trained on high-dim. sparse "hand-crafted" feature spaces may be replaced by deep NN with great success
- deep NLP vs. "traditional" NLP: advantages (many), disadvantages (some), and pitfalls (many)
- second half of lecture will exclusively deal with deep NLP:
  - part A: introduction & short concise repetition of basic NN techniques (RNNs, CNNs, backprop incomputation graphs etc.)
  - based on [1] and lectures 1-9 of [2] and lectures 1-8 of [10]

Deep NLP

- Straightforward Feed Forward NNs (FF-NNs) successfully applied in:
  - syntactic and dependency parsing
  - sub-tasks of machine translation
  - language modelling
  - sentiment classification
  - factoid question answering
  - …

- Convolutional NNs (CNNs) (using conv. layers + pooling layers) successfully applied for tasks where local clues strongly indicate class membership irrespective of their precise location in input:
  - document classification
  - sentiment classification
  - relation type classification between entities
  - event detection
  - paraphrase identification
  - question answering
  - modeling text interestingness
  - character-sequence-based POS tagging
  - …

Deep NLP

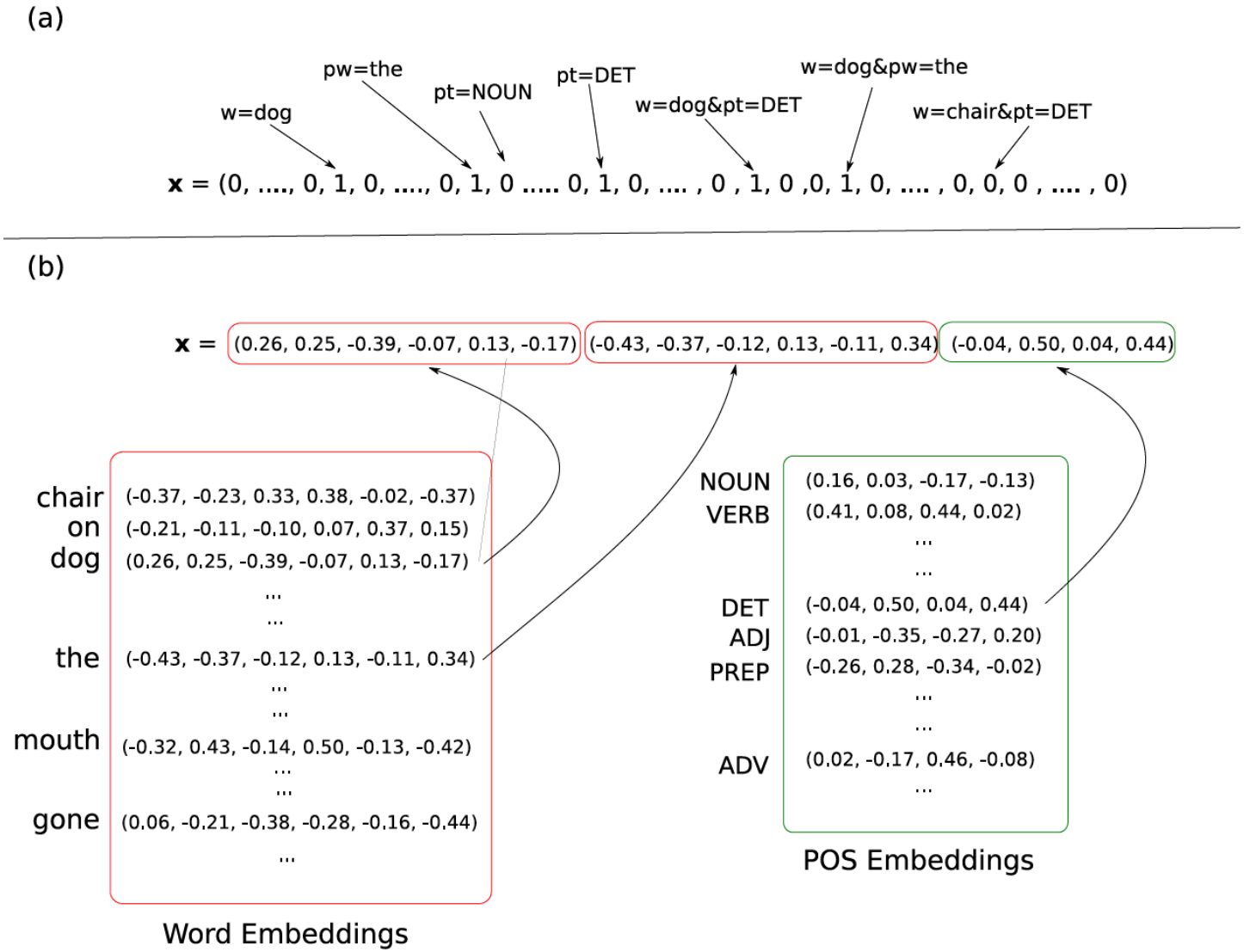
- Recurrent NNs (RNNs): CNNs ignore structural (e.g. positional) information; RNNs include this information: e.g. linear order: sequence modeling / classification; successfully applied for:
  - language modeling
  - sequence tagging
  - machine translation
  - dependency parsing
  - sentiment analysis
  - noisy text normalization
  - dialog state tracking
  - character-sequence-based POS tagging
  - …
- Recursive NNs (RecNNs): for non-linearly structured input (e.g. trees):
  - constituency parse re-ranking
  - discourse parsing
  - semantic relation classification
  - political ideology detection based on parse trees
  - sentiment classification and target-dependent sentiment classification
  - question answering
  - …

Features for Deep NLP

- NN: non-linear function NN: maps a d_{in} dim. input vector $\boldsymbol{x}$ (encoding e.g. words, POS-tags etc.) to d_{out} dim. output vector $\boldsymbol{y}$ (e.g. class probabilities)

- Traditional ML-based NLP:
  - represent each core linguistic feature $f_1, f_2, \ldots f_k$ and each combination of such features as a distinct feature space dimension → sparse, high-dim. one-hot encoded feature vectors

- Deep NLP:
  - represent each core linguistic feature f_i via dense, low-d-dim embedding vector $\boldsymbol{v}(f_i)$ (e.g. d=100 for words or d=20 for POS-tags)
  - do not initially encode feature combinations
  - train components of $\boldsymbol{v}(f_i)$ in NN
    or use pre-trained embeddings (→ embedding lookup table)
  - combine $\boldsymbol{v}(f_1)$,$\boldsymbol{v}(f_2)$,…,$\boldsymbol{v}(f_k)$ to input vector $\boldsymbol{x}$ (via concatenation, summation etc.)



Sparse vs. dense feature representations. Two encodings of the information: *current word is "dog"; previous word is "the"; previous pos-tag is "DET".*

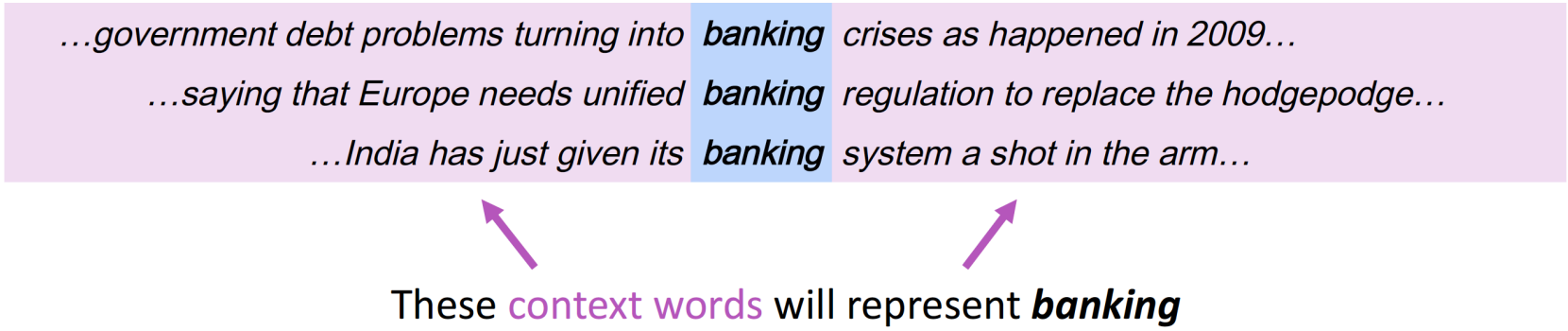
Advantages of Using Embeddings

- low dim. embeddings → low dim. inputs : better suited for NN approach

- Training embeddings ←→ similar features have similar vectors
  - → embeddings share statistical strength: **v**(cat) similar to **v**(dog)
  - → better generalization: e.g. many "dog" sentences, few "cat" sentences in training: no problem
  - for any pair of one-hot encoding vectors **e** for single words we have: $||\boldsymbol{e}(w_i) - \boldsymbol{e}(w_j)|| = \sqrt{2}$

motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
hotel = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]

Advantages of Using Embeddings



- using fixed embeddings pre-trained on large corpora: a way of transfer-learning

Feature Representation (ctd.)

- Continuous Bag of Words (CBOW)
  - usually: concatenation of embeddings of each feature as input.
    BUT: if varying number of features per training example (e.g. words occurring in document) → dimension of concatenation may vary → for fixed size input: use averaging as combination technique

$$CBOW(f_1, ..., f_k) = \frac{1}{k} \sum_{i=1}^{k} v(f_i)$$

  - variant: weighted averaging (e.g. weights == tf-idf scores)

$$WCBOW(f_1, ..., f_k) = \frac{1}{\sum_{i=1}^{k} a_i} \sum_{i=1}^{k} a_i v(f_i)$$

- Distance & Position Features
  - beneficial for e.g. event extraction: given trigger word: is a candidate word the argument? → distance matters!
  - do not use distance values as features but encode bins of distances (e.g. 1, 2, 3, 4, 5-10, 10+) via trained embeddings

Feature Representation (ctd.)

- Combinations of features
  - manual feature engineering: also represent combinations of core features ←→ enlarge feature dimension → hope for linear separability in higher dim. feature space; but: hard to do!
  - → Kernel methods:
    - no need for manually designing and explicitly computing feature vectors
    - convex optimzation (nice)
    - but: classification efficiency scales with number of training examples[1] → disadvantage for large corpora (billions of words); NN: classification efficiency scales with network size

[1] e.g. kernelized SVM, training set $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N}$: prediction:

$$y(\boldsymbol{x}) = sgn(\boldsymbol{w}^T\phi(\boldsymbol{x}) + b) = sgn\left(\left(\sum_{i=1}^{N}\alpha_i y_i \phi(\boldsymbol{x_i})\right)^T \phi(\boldsymbol{x}) + b\right) = sgn\left(\sum_{i=1}^{N}\alpha_i y_i k(\boldsymbol{x_i}, \boldsymbol{x}) + b\right);$$

Although we only require the set of support vectors $S = \{x_i | 0 < \alpha_i < C; \xi_i = 0\}$, S in principle grows with N → classification efficiency scales with N

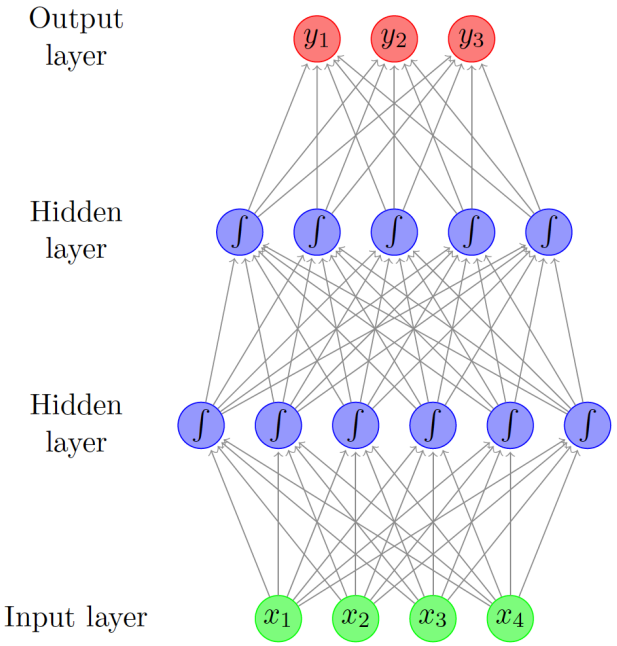
Feature Representation (ctd.)

- dimension of embeddings: determine empirically (hyper-parameter search)

- sharing of embedding vectors:
  - same embedding for w_i appearing before or after a word (**v**(dog:asNextWord) ≠ / = **v**(dog:asPreviousWord) ?)
  - same or different embedding for different word-senses?

  depends on task

Notation and Deep NN Repetition / Notation



$$NN_{Perceptron}(\mathbf{x}) = \mathbf{x}\mathbf{W} + \mathbf{b}$$
$$\mathbf{x} \in \mathbb{R}^{d_{in}},\ \mathbf{W} \in \mathbb{R}^{d_{in} \times d_{out}},\ \mathbf{b} \in \mathbb{R}^{d_{out}}$$

Simple Perceptron

$$NN_{MLP1}(\mathbf{x}) = g(\mathbf{x}\mathbf{W^1} + \mathbf{b^1})\mathbf{W^2} + \mathbf{b^2}$$
$$\mathbf{x} \in \mathbb{R}^{d_{in}},\ \mathbf{W^1} \in \mathbb{R}^{d_{in} \times d_1},\ \mathbf{b^1} \in \mathbb{R}^{d_1},$$
$$\mathbf{W^2} \in \mathbb{R}^{d_1 \times d_2},\ \mathbf{b^2} \in \mathbb{R}^{d_2}$$

One hidden layer FFNN, no output transformation

- notation in [1] : $\boldsymbol{W, A, O, W^1 = W^{(1)}}$: matrices; $\boldsymbol{b, h, x, y}$: row vectors; $[\boldsymbol{x_1}; \boldsymbol{x_2}]$: vector concatenation; W^0_{ij}: weight in weight-layer 0 from output of neuron i to input of neuron j → $\boldsymbol{h = \sigma(xW + b)}$
- notation in Goodfellow book [4]: $\boldsymbol{b, h, x, y}$: column vectors → $\boldsymbol{h = \sigma(W^{\mathrm{T}}x + b)}$
- notation in [2]: varies

$$NN_{MLP2}(\mathbf{x}) = (g^2(g^1(\mathbf{x}\mathbf{W^1} + \mathbf{b^1})\mathbf{W^2} + \mathbf{b^2}))\mathbf{W^3}$$
$$NN_{MLP2}(\mathbf{x}) = \mathbf{y}$$
$$\mathbf{h^1} = g^1(\mathbf{x}\mathbf{W^1} + \mathbf{b^1})$$
$$\mathbf{h^2} = g^2(\mathbf{h^1}\mathbf{W^2} + \mathbf{b^2})$$
$$\mathbf{y} = \mathbf{h^2}\mathbf{W^3}$$

Two hidden layers FFNN, no output transformation

NN Basics

- Universal approximation: FF-NN with single hidden layer can approximate any continuous function on compact subsets of $\mathbb{R}^n$ (under mild assumptions on the activation function and given enough hidden units). (see [5],[4])

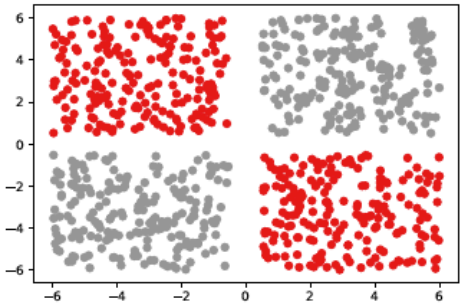
- Functions compactly represented with k layers may require exponentially many hidden units when using only k -1 layers. (see [5],[4])

- Finding the optimal weights is NP hard (see [5],[4])

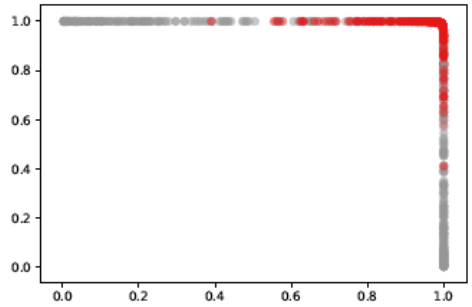
NN Basics

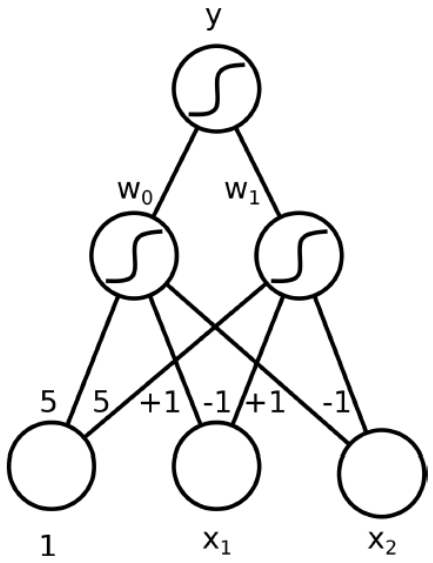
- Multiple layers: learn features of features / more specialized features / relevant feature combinations / suitable basis functions / more abstract feature representations in higher levels

$\mathbb{R}^2$ space



Transformed $\mathbb{R}^2$ space



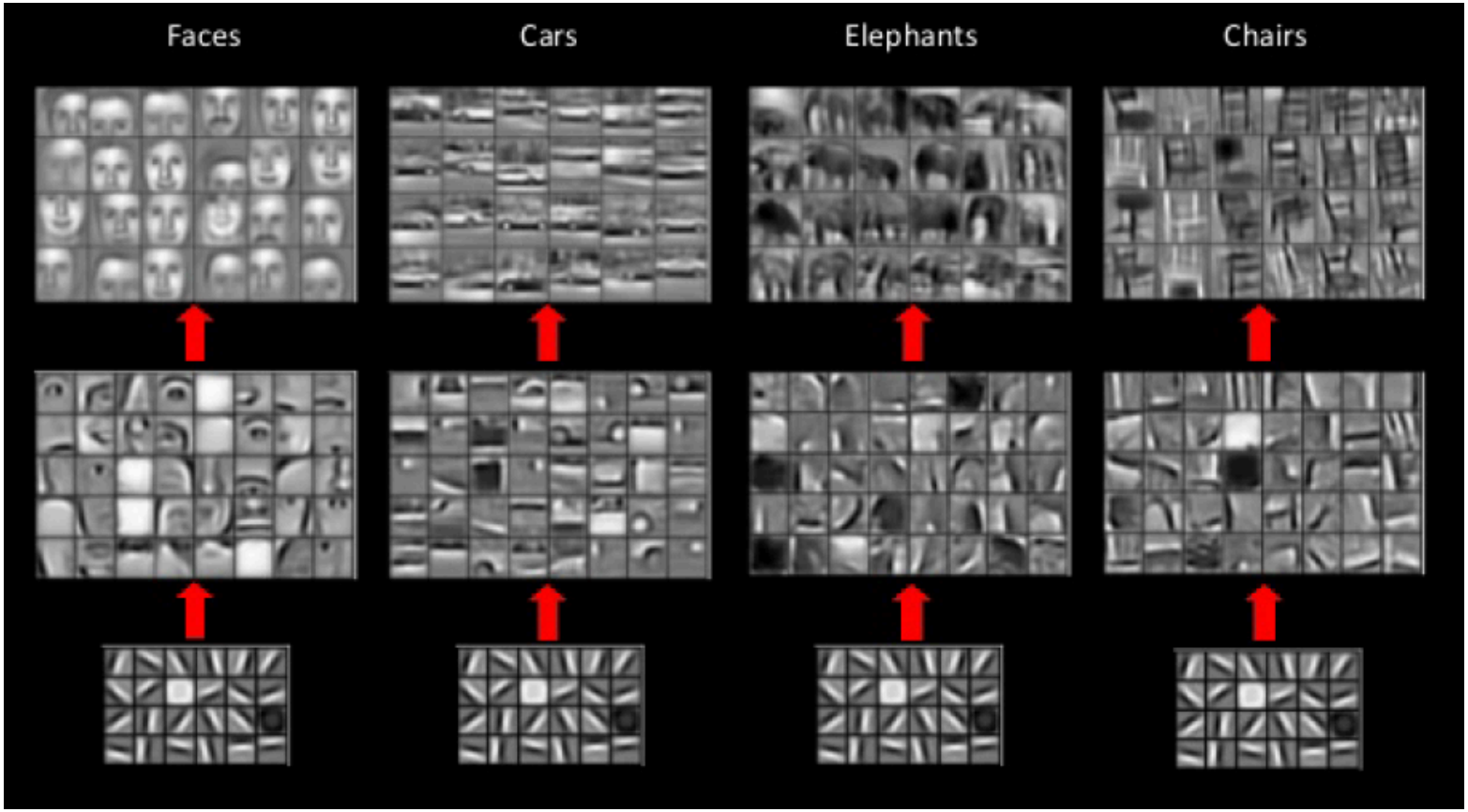


$$\boldsymbol{\phi}(\boldsymbol{x}) = \boldsymbol{\phi}(x_1, x_2) = (\ \sigma(x_1 + x_2 + 5)\ ,\ \sigma(-x_1 - x_2 + 5)\ )$$

[4],[5]

NN Basics

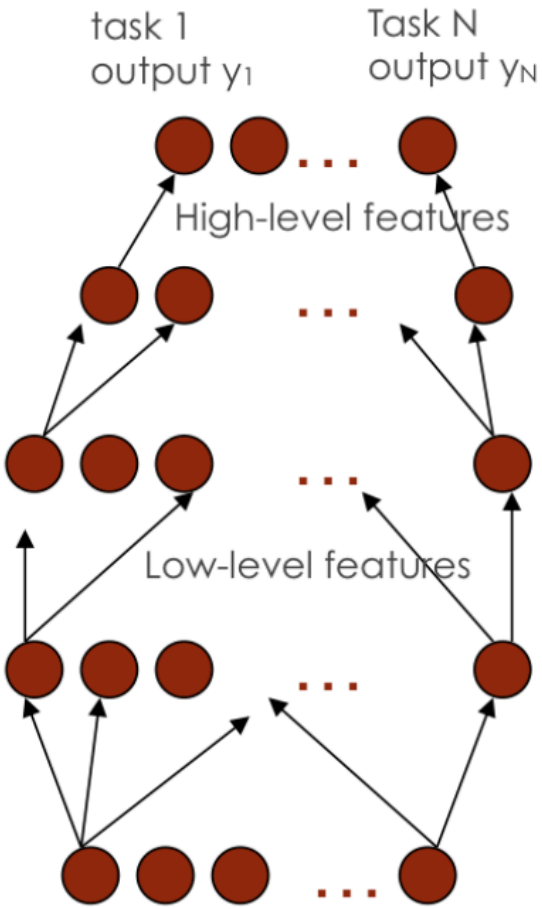
- Multiple layers: learn features of features / more specialized features / relevant feature combinations / suitable basis functions or feature representations in higher levels



[5]

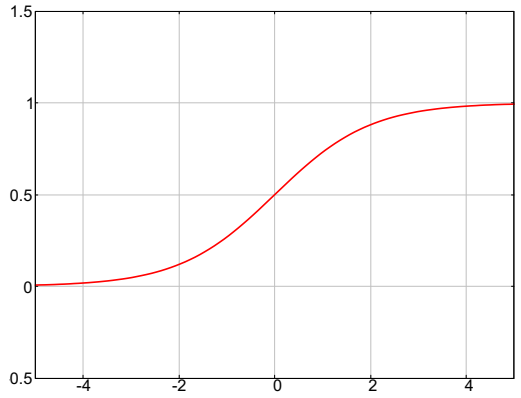
NN Basics

- Multiple layers: learn features of features / more specialized features / relevant feature combinations / suitable basis functions or feature representations in higher levels



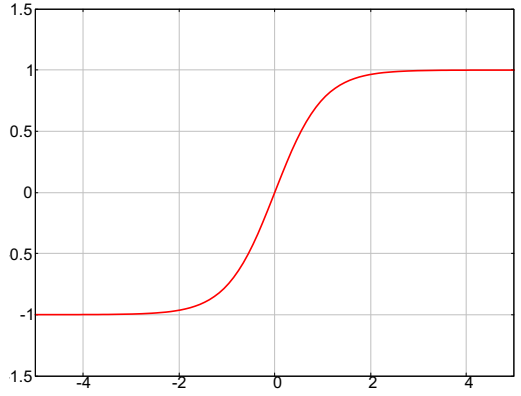
[5], [6]

Common Activation Functions / Non-Linearities



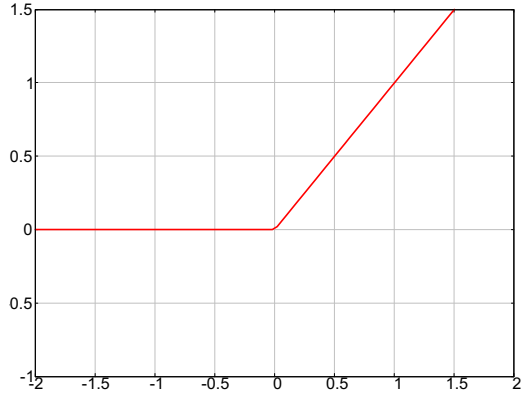
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$



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

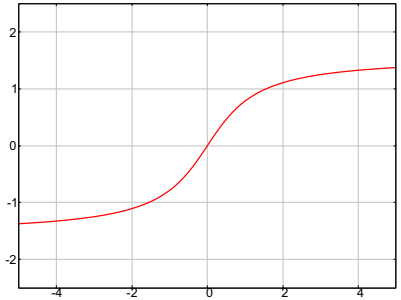
$$\tanh'(x) = 1 - \tanh(x)^2$$



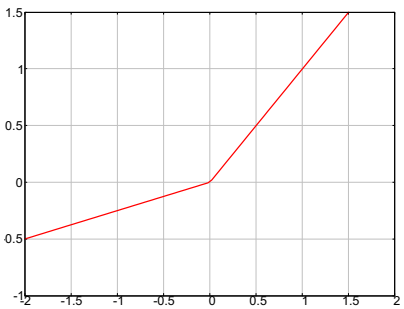
$$\text{relu}(x) = \begin{cases} x & \text{for } x \geq 0 \\ 0 & \text{for } x < 0 \end{cases}$$

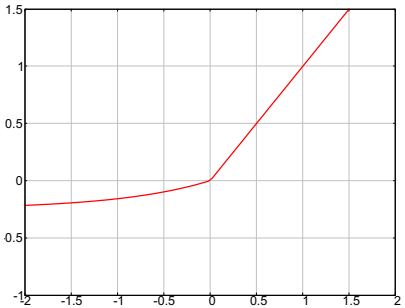
$$\text{relu}'(x) = \begin{cases} 1 & \text{for } x \geq 0 \\ 0 & \text{for } x < 0 \end{cases}$$

More Exotic Activation Functions / Non-Linearities

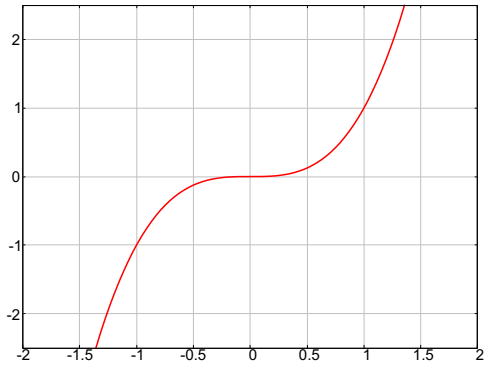


$\arctan(x)$	$\arctan'(x) = \frac{1}{1+x^2}$
$\text{pRelu}(x) = \begin{cases} x & \text{for } x \geq 0 \\ \alpha x & \text{for } x < 0 \end{cases}$	$\text{pRelu}'(x) = \begin{cases} 1 & \text{for } x \geq 0 \\ \alpha & \text{for } x < 0 \end{cases}$
$\text{elu}(x) = \begin{cases} x & \text{for } x \geq 0 \\ \alpha(e^x - 1) & \text{for } x < 0 \end{cases}$	$\text{elu}'(x) = \begin{cases} 1 & \text{for } x \geq 0 \\ \alpha + \text{elu}(x) & \text{for } x < 0 \end{cases}$





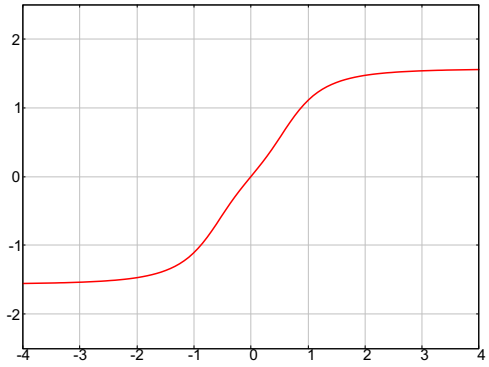
More Exotic Activation Functions / Non-Linearities



$\text{cube}(x) = x^3$

$\text{cube}'(x) = 3\text{x}^2$

both recently found to work better than other non-linearities for dependency parsing sub-tasks



$\text{tanhCube}(x) = \tanh(x^3 + x)$

$\text{tanhCube}'(x) = \dfrac{3\text{x}^2 + 1}{1 + (x^3+x)^2}$

Choice of Activation Function: Considerations

- tanh, sigmoid and other "asymptotic" functions: danger of saturated neurons → gradient very small → vanishing gradient problem
- tanh, sigmoid: derivatives involve more complicated function evaluations
- relu: no saturation, derivative is easy
- relu and other piecewise defined fcts: derivative at x=0 not defined (only subgradient)

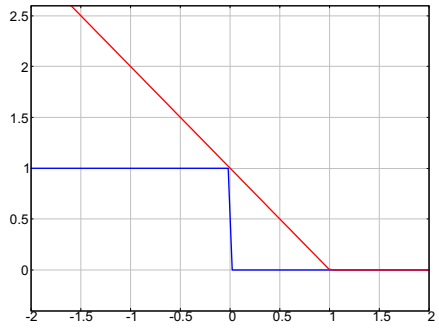
Loss Functions

- $\hat{\boldsymbol{y}} = \hat{\boldsymbol{y}}(\boldsymbol{x}, \boldsymbol{\theta})$: output of NN;
  $\boldsymbol{y} = \boldsymbol{y}(\boldsymbol{x})$: true (desired) output
  } $L(\hat{\mathbf{y}}, \mathbf{y})$

- case hard binary classification: Hinge loss (binary classification; “margin loss”, ”SVM-loss”)

$y \in \{-1,1\}$; $y^{predicted} = sgn(\hat{y})$; classification correct if $y\,\hat{y} > 0$

$$L_{hinge(binary)}(\hat{y}, y) = \max(0, 1 - y \cdot \hat{y})$$



Loss Functions

- case hard classification: Hinge loss (multi-class):

  $\boldsymbol{y}$: one hot vector (M classes)[1]; $y^{predicted} = argmax_i \hat{y}_i$;
  correct class: $t = argmax_i\ y_i$; next highest scoring class: $k = argmax_{i \neq t} \hat{y}_i$;

$$L_{hinge(multiclass)}(\hat{\mathbf{y}}, \mathbf{y}) = \max(0, 1 - (\hat{y}_t - \hat{y}_k))$$

  attempts to score the correct class above all other classes with a margin of at least 1

- case binary probabilistic classification: binary cross entropy loss

  $y \in \{0,1\}$; $\hat{y}$: e.g. sigmoid on output activation (←→ logistic regression);
  Bernoulli-model: $p(y|\boldsymbol{x}, \boldsymbol{\theta}) = \hat{y}(\boldsymbol{x}, w)^y (1 - \hat{y}(\boldsymbol{x}, \boldsymbol{\theta}))^{1-y}$ →

$$L_{crossEntBinary}(y, \hat{y}) = y \log \hat{y}(\boldsymbol{x}, \boldsymbol{\theta}) + (1 - y) \log(1 - \hat{y}(\boldsymbol{x}, \boldsymbol{\theta}))$$

[1] in [1] the number of classes is denoted as n, which we avoid here to avoid confusion with the usual use of n as an index of the training examples

Loss Functions

- case probabilistic multi-class classification: cross entropy loss

hard (crisp, exclusive) classification

$\boldsymbol{y}$: one hot vector (M classes); $\hat{\boldsymbol{y}}$: softmax on output activations;
categorical-(multinoulli)-model: $p(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\theta}) = \prod_{i=1}^{M} \hat{y}(\boldsymbol{x},\boldsymbol{\theta})_i{}^{y_i}$ →

$$L_{crossEntropy}(\boldsymbol{y},\hat{\boldsymbol{y}}) = -\sum_{i=1}^{M} y_i \log(\hat{y}(\boldsymbol{x},\boldsymbol{\theta})_i)$$

$$= -\log(\hat{y}(\boldsymbol{x},\boldsymbol{\theta})_t$$

non-exclusive classification:

Again: $\hat{\boldsymbol{y}}$: softmax on output activations;
Assume that in the training data-set $\{(\boldsymbol{x}_n,\boldsymbol{y}_n)\}_{n=1}^{N}$ one $\boldsymbol{x}$ occurs Q times, with a discrete count "distribution" $\boldsymbol{y}$ with $\sum_{i=1}^{M} y_i = Q$. So each of the Q iid $\boldsymbol{x}$ has a categorical model $p(\tilde{\boldsymbol{y}}|\boldsymbol{x},\boldsymbol{\theta}) = \prod_{i=1}^{M} \hat{y}(\boldsymbol{x},\boldsymbol{\theta})_i{}^{\tilde{y}_i}$, where $\tilde{\boldsymbol{y}}$ is a one-hot vector.
For all Q iid $\boldsymbol{x}$ together this gives $p(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\theta}) = \prod_{q=1}^{Q}\prod_{i=1}^{M} \hat{y}(\boldsymbol{x},\boldsymbol{\theta})_i{}^{\tilde{y}_i} = \prod_{i=1}^{M} \hat{y}(\boldsymbol{x},\boldsymbol{\theta})_i{}^{y_i}$ which, neglecting the multinomial coefficient, is a multinomial expression.
→ $L_{crossEntropy}(\boldsymbol{y},\hat{\boldsymbol{y}}) = -\sum_{i=1}^{M} y_i \log(\hat{y}(\boldsymbol{x},\boldsymbol{\theta})_i)$. If we normalize $\boldsymbol{y}$ by diving by Q , we thus essentially match the NN output distribution with a desired per-$\boldsymbol{x}$-distribution $\boldsymbol{y}$. (regard that we have $L_{crossEntropy}(\boldsymbol{y},\hat{\boldsymbol{y}}) = \mathrm{H}(\boldsymbol{y}) - \mathrm{KL}(\boldsymbol{y}||\hat{\boldsymbol{y}})$

Loss Functions

- Ranking Hinge loss:

given set of correct items $\boldsymbol{x}$ and incorrect items $\boldsymbol{x'}$(usually constructed from $\boldsymbol{x}$) :
goal: rank the $\boldsymbol{x}$ higher than the $\boldsymbol{x'}$.
(e.g. rank correct verb-object pairs above incorrect, automatically derived ones, or rank correct (head,relation,trail) triplets above corrupted ones in an information-extraction setting

$$L_{ranking(margin)}(\boldsymbol{x}, \boldsymbol{x}') = \max(0, 1 - (\hat{\boldsymbol{y}}(\boldsymbol{x}, \boldsymbol{\theta}) - \hat{\boldsymbol{y}}(\boldsymbol{x}', \boldsymbol{\theta})))$$

$$L_{ranking(log)}(\boldsymbol{x}, \boldsymbol{x}') = \log(1 + \exp(-(\hat{\boldsymbol{y}}(\boldsymbol{x}, \boldsymbol{\theta}) - \hat{\boldsymbol{y}}(\boldsymbol{x}', \boldsymbol{\theta}))))$$

Embeddings

- Using pre-trained embeddings and a lookup table E .
  e.g. concatenating embedding vectors:

$$\begin{aligned}\mathbf{x} = c(f_1, f_2, f_3) =& [v(f_1); v(f_2); v(f_3)] \\ NN_{MLP1}(\mathbf{x}) =& NN_{MLP1}(c(f_1, f_2, f_3)) \\ =& NN_{MLP1}([v(f_1); v(f_2); v(f_3)]) \\ =& (g([v(f_1); v(f_2); v(f_3)]\mathbf{W^1} + \mathbf{b^1}))\mathbf{W^2} + \mathbf{b^2}\end{aligned}$$

c: function from core features to input vector

with f_i one hot core features (formally; (in practice often efficiently implement lookup step with hash table):

$$v(f_i) = \mathbf{f_i E}$$

or summing embedding vectors

$$\begin{aligned}\mathbf{x} = c(f_1, f_2, f_3) =& v(f_1) + v(f_2) + v(f_3) \\ NN_{MLP1}(\mathbf{x}) =& NN_{MLP1}(c(f_1, f_2, f_3)) \\ =& NN_{MLP1}(v(f_1) + v(f_2) + v(f_3)) \\ =& (g((v(f_1) + v(f_2) + v(f_3))\mathbf{W^1} + \mathbf{b^1}))\mathbf{W^2} + \mathbf{b^2}\end{aligned}$$

Embeddings

- or learn embeddings:
  in its simplest form this is very similar to the previous approach: dedicate first layer to learn “embeddings”:
  assume dim(feature space) = $|V|$; case k core one-hot (sparse) feature vectors $\mathbf{f_i}$: the first layer would be:

$$\mathbf{xW} + \mathbf{b} = (\sum_{i=1}^{k} \mathbf{f_i})\mathbf{W} + \mathbf{b}$$

$$\mathbf{W} \in \mathbb{R}^{|V| \times d}, \quad \mathbf{b} \in \mathbb{R}^{d}$$

(this is very similar to CBOW using pre-trained embeddings in lookup table $\mathbf{E}$:

$$CBOW(f_1, ..., f_k) = \sum_{i=1}^{k} (\mathbf{f_i E}) = (\sum_{i=1}^{k} \mathbf{f_i})\mathbf{E}$$

differences: bias **b,** activation function of embedding learning layer, and the possibility to share embeddings between certain f_i (e.g. “next word is dog” , “previous word is dog”) in $\mathbf{E}$)

Word Embeddings: Initialization & Supervised Pre-Training

- learn d-dim. word embedding vectors: initialization:
  - random (mainly for frequent core features (e.g. POS tags, characters)):
    - sample from uniform distribution in $[-\frac{1}{2d}, \frac{1}{2d}]$ (word2vec)
    - or $[-\frac{\sqrt{6}}{\sqrt{d}}, \frac{\sqrt{6}}{\sqrt{d}}]$ (Xavier initialization)
  - or initialize with pre-trained embeddings (mainly for rare core features (e.g. words))

- supervised pre-training (cross task transfer learning):
  pre-train word embeddings on auxiliary task B (where more labeled data is available (e.g. for POS tagging)) and use them for goal task A (less labeled data available)

Word Embeddings: Unsupervised Pre-Training

- usually: never enough labelled data → use auxiliary task B on very large text corpora and unsupervised methods for training
- advantage: rare words (appearing only in test data for actual task but not in training data for actual task) appear in large corpus in similar contexts as in test data for actual task → sharing statistical strength, better generalization
- goal: similar words have similar embeddings
  (←→ distributional hypothesis (Harris, 1954): words are similar if they appear in similar contexts.)
- similarity: hard to define, task dependent, syntactic vs semantic etc.

"You shall know a word by the company it keeps" (Firth 1957)

Word Embeddings: Unsupervised: Training Objectives

word w, context c, $v(w)$: d-dim. randomly initialized embedding vector

- language modelling inspired auxiliary task:

  predict w given c ←→ learn $p(w|c)$ (word2vec, GloVe)

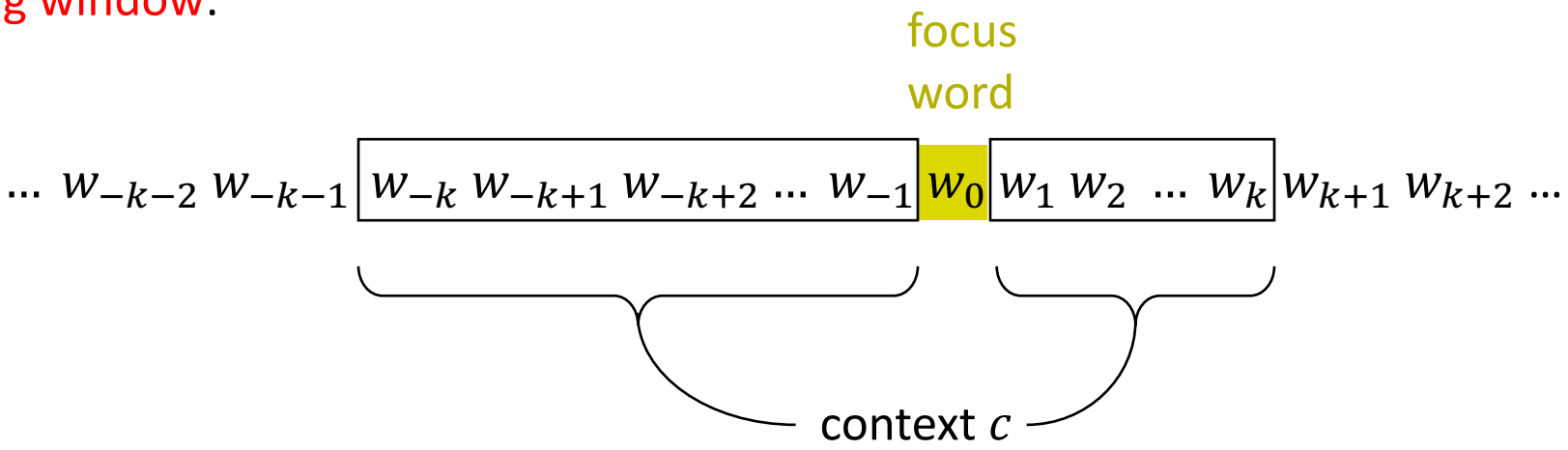
- auxiliary task: binary classification

  btw. $\mathrm{D} = \{(w,c) \in \text{corpus}\}$ and $\overline{\mathrm{D}} = \{\text{random } (w,c) \notin \text{corpus}\}$ e.g. using
  - margin-based binary ranking approach + FF-NN or
  - probabilistic models $p((w,c) \in \mathrm{D} \mid (w,c))$ + NN

tendency: choice of auxiliary task and context has more influence on embedding usefulness for actual task than exact learning method

Word Embeddings: Choice of Context

sliding window:



variants: predicting:

- $w_0 \mid CBOW(v(w_{-k}), v(w_{-k+1}), \dots, v(w_{-1}), v(w_1), \dots, v(w_k))$
- $w_0 \mid [v(w_{-k}), v(w_{-k+1}), \dots, v(w_{-1}), v(w_1), \dots, v(w_k)]$
- 2k single tasks (skip gram model):
  $w_0 \mid v(w_{-k})$ and $w_0 \mid v(w_{-k+1}), \dots,$ and $w_0 \mid v(w_k)$
  (or vice versa: $v(w_{-k}) \mid w_0$ etc.)

Word Embeddings: Effect of Window Size

- smaller windows: tend to capture functional and syntactic relations and narrower semantic relatedness
  → similar embeddings for

  Poodle, Pitbull, Rottweiler or

  walking, running, approaching

- larger windows: tend to capture topical / wider semantic relatedness
  → similar embeddings for

  dog, bark, leash

  walked, run, walking

focus word	CBOW, k=5	CBOW, k=2
batman	nightwing	superman
	aquaman	superboy
	catwoman	aquaman
	superman	catwoman
	manhunter	batgirl
hogwarts	dumbledore	evernight
	hallows	sunnydale
	half-blood	garderobe
	malfoy	blandings
	snape	collinwood
turing	nondeterministic	non-deterministic
	non-deterministic	finite-state
	computability	nondeterministic
	deterministic	buchi
	finite-state	primality
florida	gainesville	fla
	fla	alabama
	jacksonville	gainesville
	tampa	tallahassee
	lauderdale	texas
object-oriented	aspect-oriented	aspect-oriented
	smalltalk	event-driven
	event-driven	objective-c
	prolog	dataflow
	domain-specific	4gl
dancing	singing	singing
	dance	dance
	dances	dances
	dancers	breakdancing
	tap-dancing	clowning

[7]

Word Embeddings: Further Variants of Window Approach

- additionally using positional information of words in window: together with smaller windows → more syntactic similarities, group together words sharing POS + functionally similar in terms of semantics.

- further variations:
  - word lemmatization before learning
  - text normalization
  - filter too short or too long sentences
  - remove capitalization
  - sub-sampling: skipping with some probability windows that have too common or too rare focus words.
  - dynamic window size
  - weight positions in window differently (e.g. closer to focus word = higher weight)

Word Embeddings: Further Variants of Window Approach

- stuctural units (sentences, paragraphs, sections, documents, table entries etc.) as context

- replace linear context (window) with syntactic context (e.g. based on proximity in dependency parse tree + common parse relation)
  - → functional similarities, grouping together words than can fill the same role in a sentence (e.g. colors, names of schools, verbs of movement).
  - → syntactic grouping , grouping together words that share an inflection

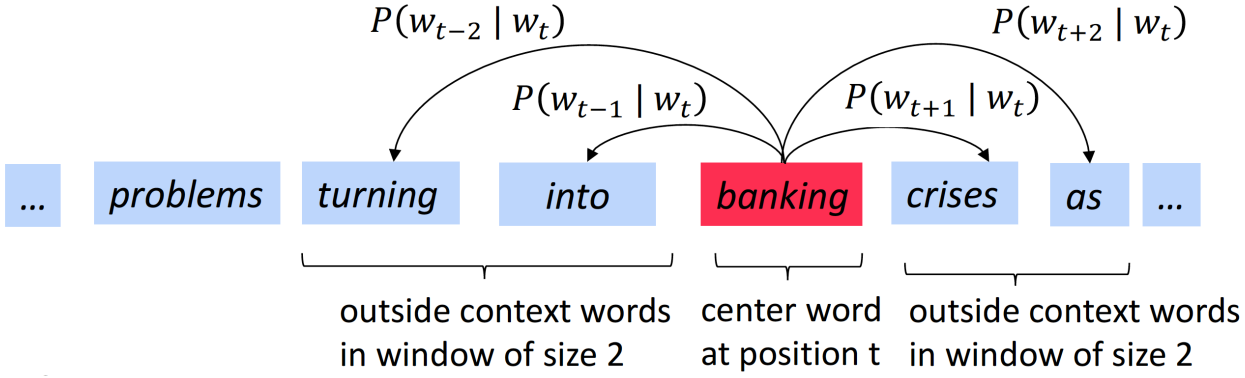
- multi-lingual: use bilingual sentence aligned corpus, align words / phrases with alignment model, use foreign word context for focus word (→ group synonyms) or mixed language contexts for focus word (→ reduce embedding similarity of antonyms (*hot, cold*)

Character Embeddings

- → smaller embedding matrices E
- → rather syntactic similarities
- good results: CNN approaches as well as Bi-LSTM approaches for POS tagging or dependency parsing using character embeddings
- allows for computing embeddings for unseen words
- however: character embeddings are challenging: relationship between form (characters) and function (syntax, semantics) in language is quite loose.
- also possible: combine usual word-embeddings with character-based word-embeddings, or sub-word-based embeddings, or other morphological element embeddings:
  allows for back-off in case of unseen words.

word2Vec vs GloVe: word2Vec

word2vec
basic approach:
skip-gram

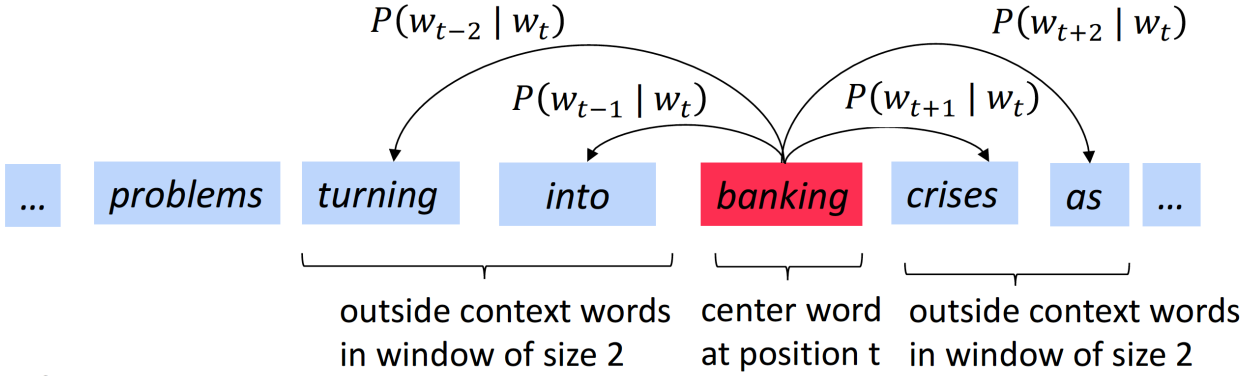


$$L(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \le j \le m \\ j \neq 0}} P(w_{t+j} \mid w_t; \theta)$$

$$J(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \le j \le m \\ j \neq 0}} \log P(w_{t+j} \mid w_t; \theta)$$

word2Vec vs GloVe: word2Vec

word2vec
basic approach:
skip-gram



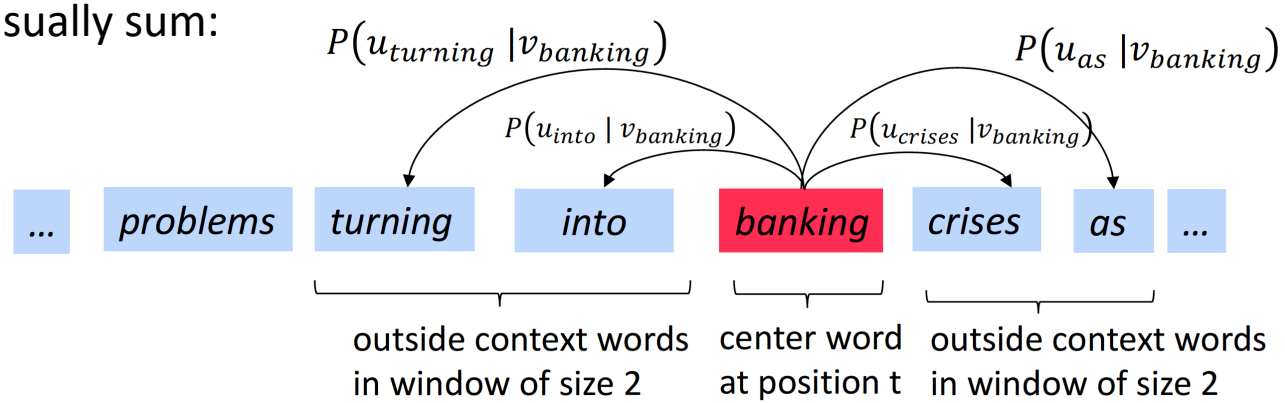
$$L(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \le j \le m \\ j \neq 0}} P(w_{t+j} \mid w_t; \theta)$$

$$J(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \le j \le m \\ j \neq 0}} \log P(w_{t+j} \mid w_t; \theta)$$

model $P(w_o|w_c)$: use two vectors per word:

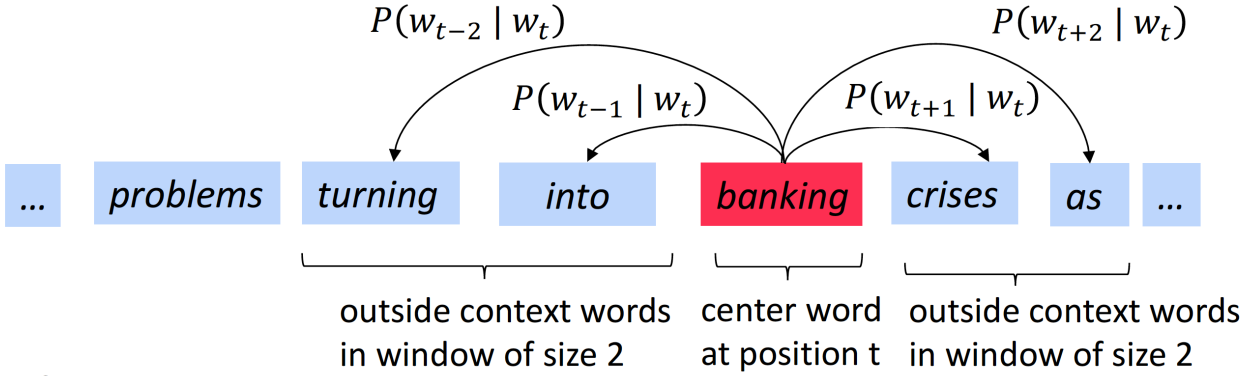
- v_w when w is center word w_c
- u_w when w is a context word w_o

final embedding for w: usually sum: $u_w + v_w$



word2Vec vs GloVe: word2Vec

word2vec
basic approach:
skip-gram



$$L(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \le j \le m \\ j \neq 0}} P(w_{t+j} \mid w_t; \theta)$$

$$J(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \le j \le m \\ j \neq 0}} \log P(w_{t+j} \mid w_t; \theta)$$

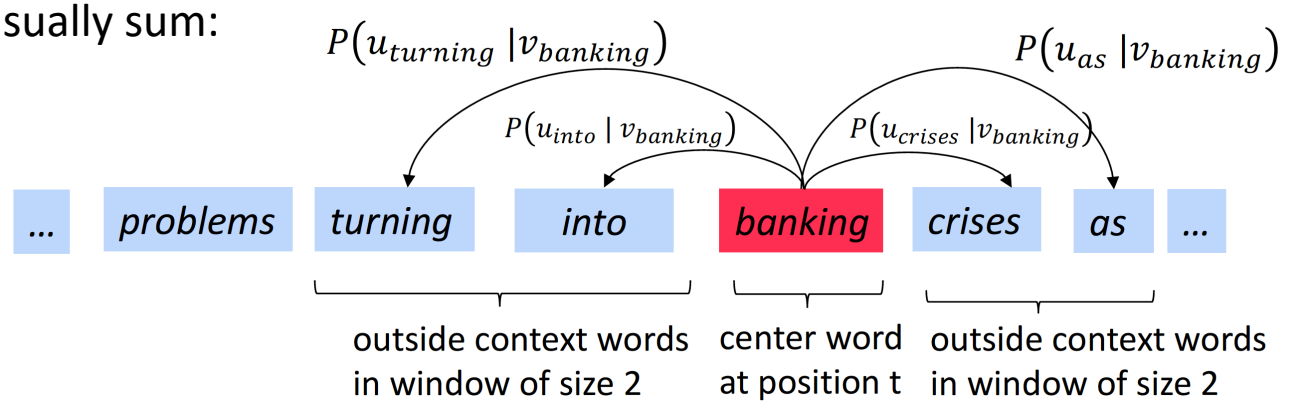
model $P(w_o|w_c)$: use two vectors per word:

- v_w when w is center word w_c
- u_w when w is a context word w_o

final embedding for w: usually sum: $u_w + v_w$

naive way to compute $P(w_o|w_c)$:
(o: outside = context); c = center) softmax:

$$P(w_o|w_c) = \frac{\exp(u_o^T v_c)}{\sum_{o' \in V} \exp(u_{o'}^T v_c)}$$



word2Vec vs GloVe: word2Vec

better way: use negative sampling, just sample one u_o from the window, and use $P(w_o|w_c) = \sigma(u_o^T v_c)$:

$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J_t(\theta)$$

$$J_t(\theta) = -\left(\log \sigma(u_o^T v_t) - \sum_{n=1}^{k} \mathbb{E}_{n\sim P(w)}\left[\log(-u_n^T v_t)\right]\right)$$

$$\approx -\log \sigma(u_o^T v_t) - \sum_{n=1}^{k} \log \sigma(-u_n^T v_t)$$

where we sample k negative samples from modified unigram distributions:

$$\mathrm{j} \sim P(w) = \frac{1}{Z} U(w)^{\frac{3}{4}}$$

→ maximize the positive element (from actual context word) while minimize the k elements from the negative (random) samples

word2Vec vs GloVe: word2Vec

- → two weight matrices $\boldsymbol{U}, \boldsymbol{V} \in \mathbb{R}^{|\boldsymbol{V}| \times \boldsymbol{d}}$
  → embedding matrix $\boldsymbol{E} = \boldsymbol{U} + \boldsymbol{V} \in \mathbb{R}^{|\boldsymbol{V}| \times \boldsymbol{d}}$

- word2vec:
  - **disadvantage**: captures co-occurrence of words one at a time;
  - **alternative**: use co-occurrence count matrix (either window-based or document-based co-occurrence counts) + truncated SVD or PCA for dimensionality reduction → Latent Semantic Analysis (LSA) (see earlier chapters)

count-based: LSA	direct prediction: word2Vec
+ efficient usage of co-occurrence statistics ○ captures "topical" word similarities – disproportionate importance given to large co-occ-counts with common words (→ limit counts, stop-word removal, use of positive Pearson correlations only, use PPMI etc.) – computational cost scales quadratically in $\|V\|$ – hard to incorporate new words or docs – not NN learning paradigm	– inefficient usage of global co-occurrence statistics – scales with corpus size + easier to incorporate new words + NN style learning → easy to combine with other NN tasks + can capture complex beyond "topical" word-similarities (e.g. depending on window size) + proven to enhance performance for other target tasks

word2Vec vs GloVe: GloVe

● , [3]

- idea: in "separate" dimensions of embedding vector space (←→ distance, angle) capture various aspects of semantic relatedness, using statistical strength of co-occurrence matrix

- definitions:
  - co-occurrence matrix: X_{ij}: number of times word j occurs in context of word i.
  - $X_i = \sum_k X_{ik}$: number of times any word appears in context of word i.
  - $P_{ij} = P(j|i) = X_{ij}/X_i$: probability that word j appear in context of word i

Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96
	ratio large → *solid* occurs with *ice* but not with *steam* → "specific" for *ice*	ratio small → *gas* occurs with *steam* but not with *ice* → "specific" for steam	ratio close to 1 → *water* occurs equally often with *steam* as with *ice* → "specific" for neither of them	ratio close to 1 → *fashion* occurs equally rarely with *steam* as with *ice* → "specific" for neither of them

word2Vec vs GloVe: GloVe

● , [3]

- → assumption: model for embeddings should depend on ratios of probabilities, employing a function F of "center" embeddings $u \in \mathbb{R}^d$ and context embeddings $v \in \mathbb{R}^d$:

$$F(u_i, u_j, v_k) = \frac{P_{ik}}{P_{jk}}$$

- retain "linear structure" of embedding vector space (e.g. in view of "equations" such as $king - queen = man - woman$) → further assumption:

$$F\left(\left(u_i - u_j\right)^T v_k\right) = \frac{P_{ik}}{P_{jk}}$$

- desired: symmetry regarding exchanging $u \leftrightarrow v$ (and $X \leftrightarrow X^T$ (fulfilled anyway for symmetric contexts)): achieved by setting:

$$F\left(\left(u_i - u_j\right)^T v_k\right) = F\left(u_i^T v_k - u_j^T v_k\right) = \frac{F(u_i^T v_k)}{F(u_j^T v_k)}$$

↔ F is a homomorphism between the groups $(\mathbb{R}, +)$ and $(\mathbb{R}, \times)$

word2Vec vs GloVe: GloVe

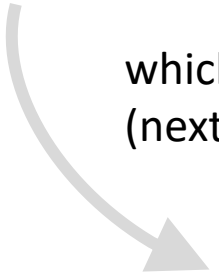
● , [3]

which is solved by setting $F\left(u_i^T v_k\right) = P_{ik}$ and $F = exp$ so that

$$u_i^T v_k = \log(P_{ik}) = \log(X_{ik}) - \log(X_i) := \log(X_{ik}) - b_i$$

so finally adding a constant b_k we achieve $u \leftrightarrow v$ symmetry:

$$u_i^T v_k + b_i + b_k = \log(X_{ik})$$



which leads to
(next slide)

word2Vec vs GloVe: GloVe

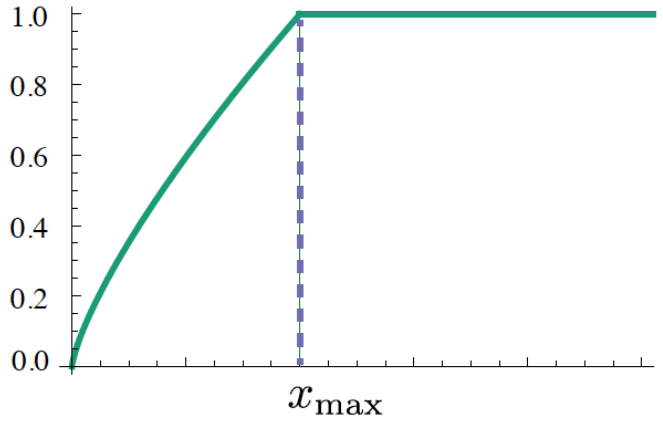
● , [3]

→ heuristically define weighted least squares model:

$$J = \sum_{i,j=1}^{|V|} f(X_{ij})\,(u_i^T v_j + b_i + b_j - \log(X_{ij}))^2$$

where weighting function f should have the following properties:

- $f(0) = 0$ with an asymptotic behaviour that is "compatible" with the other "ingredients" of the model to ensure mathematical "well-behavedness" such that 0 log 0 = 0 (zero co-occurrences do not matter)
- f should be non-decreasing to not over-emphasize rare co-occurrences (small P_{ij})
- f should be not too large for frequent "noisy" co-occurrences (large P_{ij})



final embedding after training: $x = u + v$

word2Vec vs GloVe: Relationship

● , [3]

prediction-based models: (theoretical, but because of denominator impractical) embedding-based probability that word j appears in context of word i:

$$P(w_j|w_i) := Q_{ij} = \frac{\exp(u_i^T v_j)}{\sum_{j' \in V} \exp(u_i^T v_j)}$$

→ loss (without neg. sampling):

$$J = -\sum_{\substack{i \in \text{corpus} \\ j \in \text{context}(i)}} \log Q_{ij} = -\sum_{i=1}^{V} \sum_{j=1}^{V} X_{ij} \log Q_{ij}$$

$$= -\sum_{i=1}^{V} X_i \sum_{j=1}^{V} P_{ij} \log Q_{ij} = \sum_{i=1}^{V} X_i H(P_i, Q_i)$$

where

- $X_i = \sum_k X_{ik}$: number of times any word appears in context of word i.
- $P_{ij} = P(i|j) = X_{ij}/X_i$: co-occurrence-based probability that word j appear in context of word i
- P_i: co-occurrence-based probability that any word appears in context of word i
- Q_i: embedding-based probability that any word appears in context of word i
- $H(P_i, Q_i)$: cross entropy

word2Vec vs GloVe: Relationship

● , [3]

- cross entropy: too much weight for unlikely events → replace the cross entropy with a squared error loss function →

$$J = \sum_{ij} X_i \left(\hat{P}_{ij} - \hat{Q}_{ij}\right)^2$$

where $\hat{P}_{ij} = X_{ij}$ and $\hat{Q}_{ij} = \exp(u_i^T v_j)$ are the unnormalized distributions

- X_{ij} too large values → replace with log

$$J = \sum_{ij} X_i \left(\log \hat{P}_{ij} - \log \hat{Q}_{ij}\right)^2 = \sum_{ij} X_i \left(u_i^T v_j - \log X_{ij}\right)^2$$

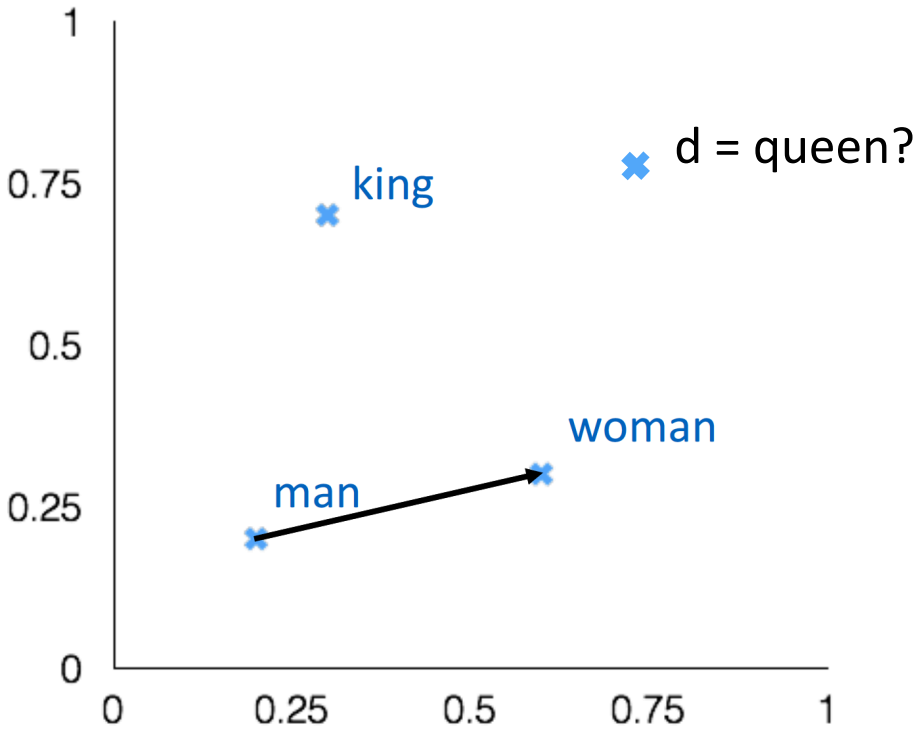
- use a more general weighting function (+ treat u, v as absorbing the biases b) →

$$J = \sum_{i,j=1}^{|V|} f\left(X_{ij}\right) \left(u_i^T v_j - \log\left(X_{ij}\right)\right)^2$$

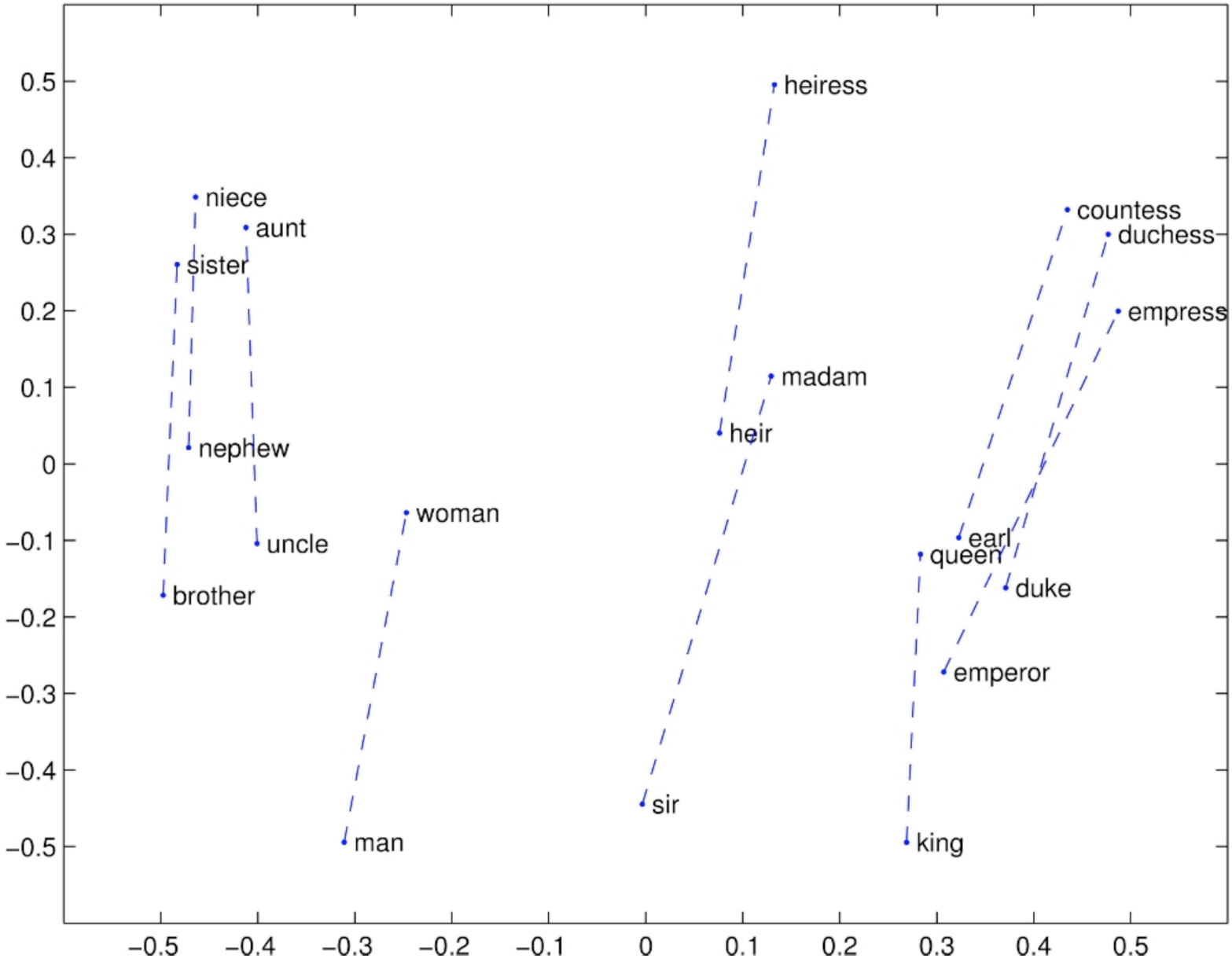
word2Vec vs GloVe: Intrinsic Evaluation: Analogy Task

- how well do vector addition / subtraction plus inner product (cosine) capture syntactic + semantic analogies:
  a : b :: c : d?

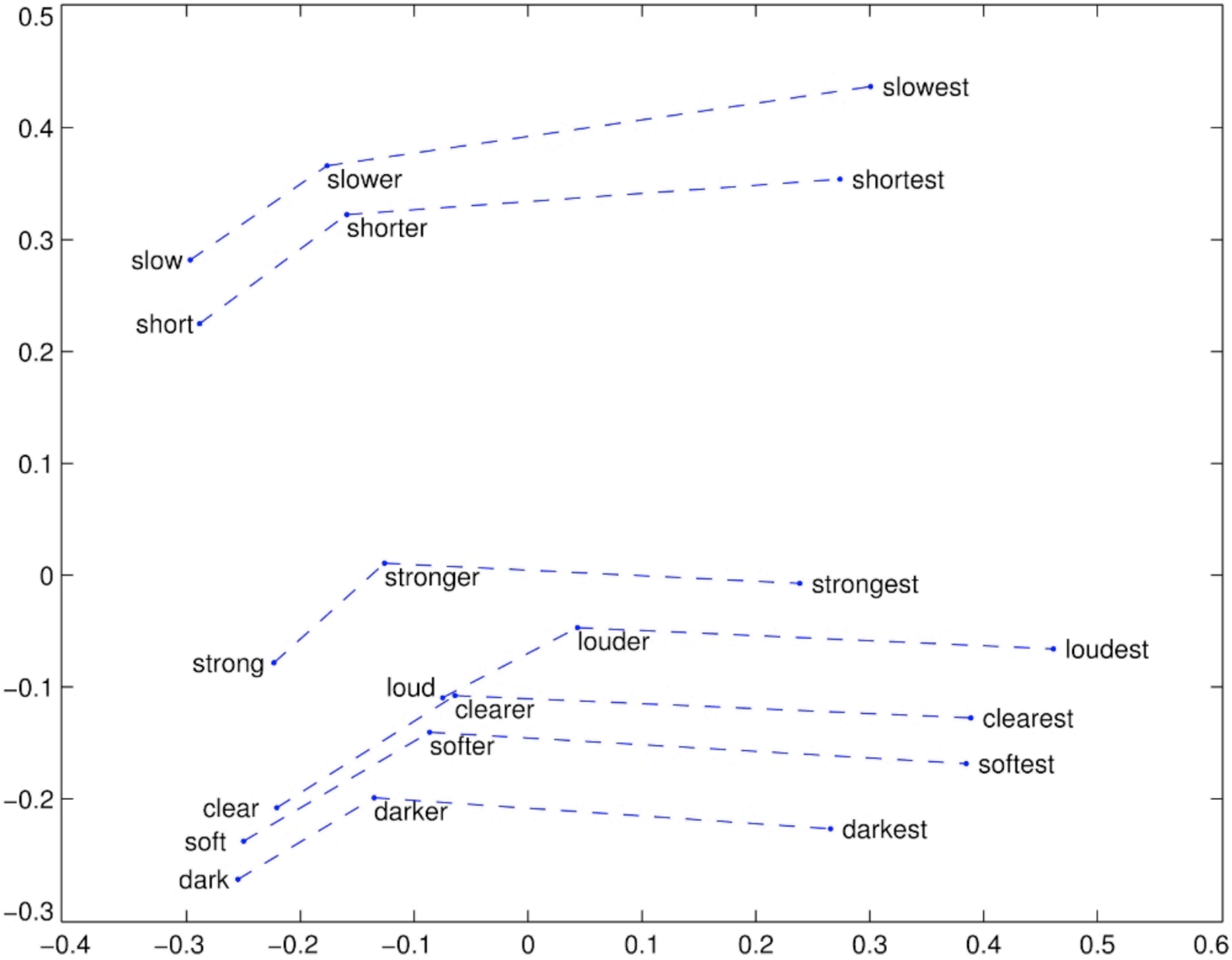
$$d = \arg\max_i \frac{(x_b - x_a + x_c)^T x_i}{||x_b - x_a + x_c||}$$



word2Vec vs GloVe: Glove Examples



word2Vec vs GloVe: Glove Examples - Superlatives



word2Vec vs GloVe: word2Vec Examples – Analogy Task

Expression	*Nearest token*
Paris - France + Italy	Rome
bigger - big + cold	colder
sushi - Japan + Germany	bratwurst
Cu - copper + gold	Au
Windows - Microsoft + Google	Android
Montreal Canadiens - Montreal + Toronto	Toronto Maple Leafs

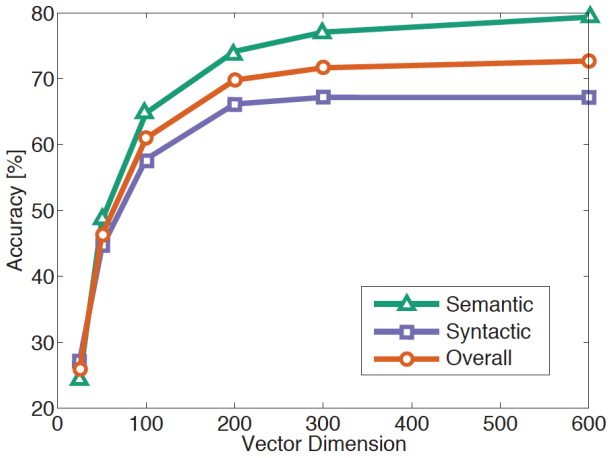
word2Vec vs GloVe: Analogy Task: Performance

● , [3]

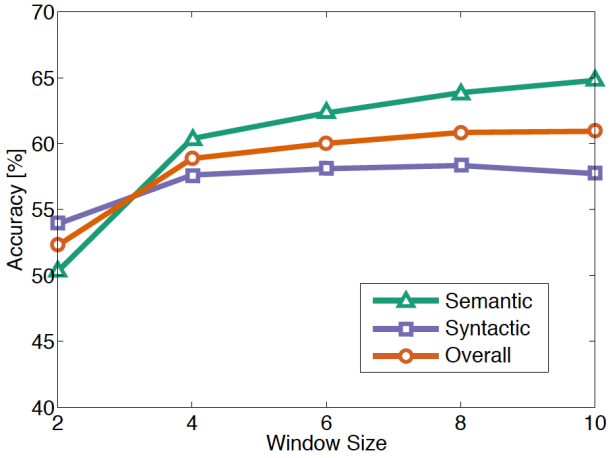
- dataset by Mikolov et al (2013): 19544 analogy questions:
  - semantic: e.g. Athens : Greece :: Berlin : ?
  - syntactic: e.g. dance : dancing :: fly : ?
  - SG=word2vec (skip-gram)
  - CBOW=word2vec (CBOW)

Model	Dim.	Size	Sem.	Syn.	Tot.
ivLBL	100	1.5B	55.9	50.1	53.2
HPCA	100	1.6B	4.2	16.4	10.8
GloVe	100	1.6B	67.5	54.3	60.3
SG	300	1B	61	61	61
CBOW	300	1.6B	16.1	52.6	36.1
vLBL	300	1.5B	54.2	64.8	60.0
ivLBL	300	1.5B	65.2	63.0	64.0
GloVe	300	1.6B	80.8	61.5	70.3
SVD	300	6B	6.3	8.1	7.3
SVD-S	300	6B	36.7	46.6	42.1
SVD-L	300	6B	56.6	63.0	60.1
CBOW†	300	6B	63.6	67.4	65.7
SG†	300	6B	73.0	66.0	69.1
GloVe	300	6B	77.4	67.0	71.7
CBOW	1000	6B	57.3	68.9	63.7
SG	1000	6B	66.1	65.1	65.6
SVD-L	300	42B	38.4	58.2	49.2
GloVe	300	42B	**81.9**	**69.3**	**75.0**

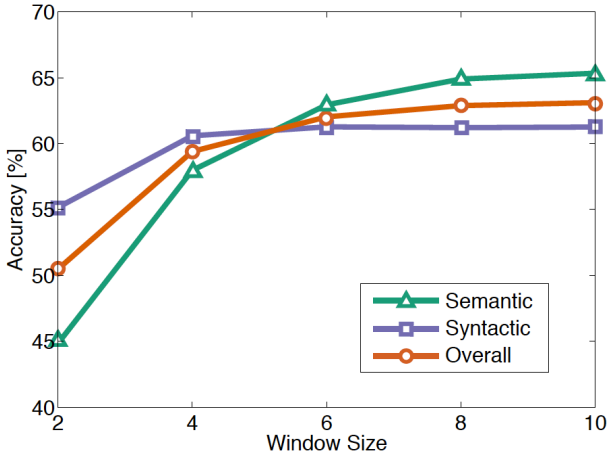
word2Vec vs GloVe: GloVe: Analogy Task: Performance ● , [3]



(a) Symmetric context



(b) Symmetric context



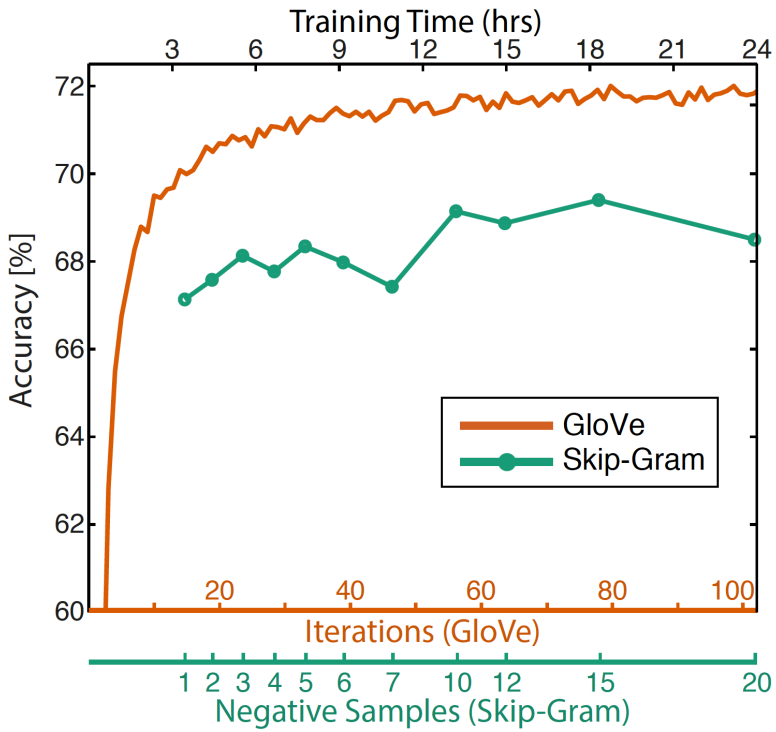
(c) Asymmetric context

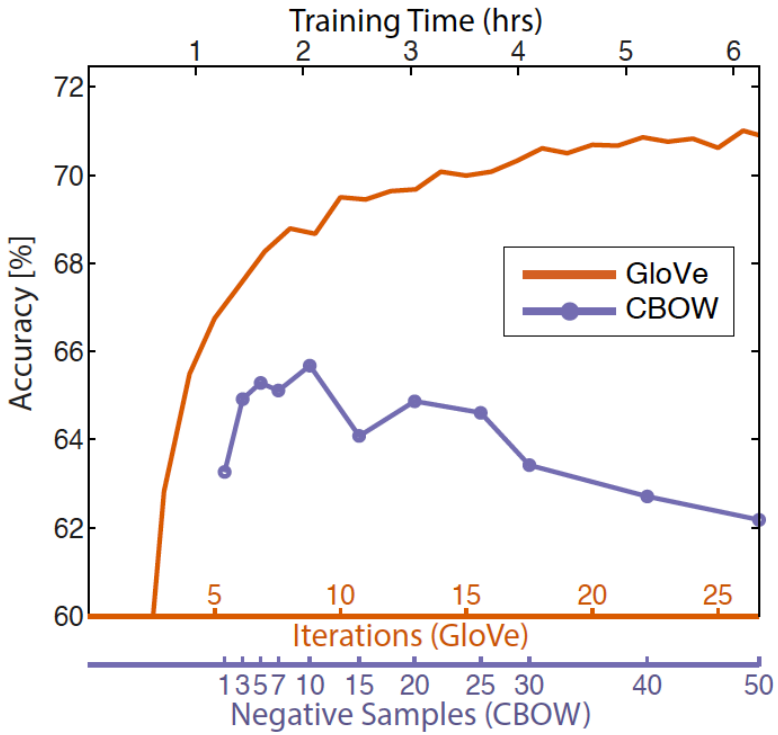
→ symmetric contexts are better
→ optimal dimension here ≈ 300
→ optimal window size here ≈ 8

word2Vec vs GloVe: Analogy Task: Performance

● , [3]

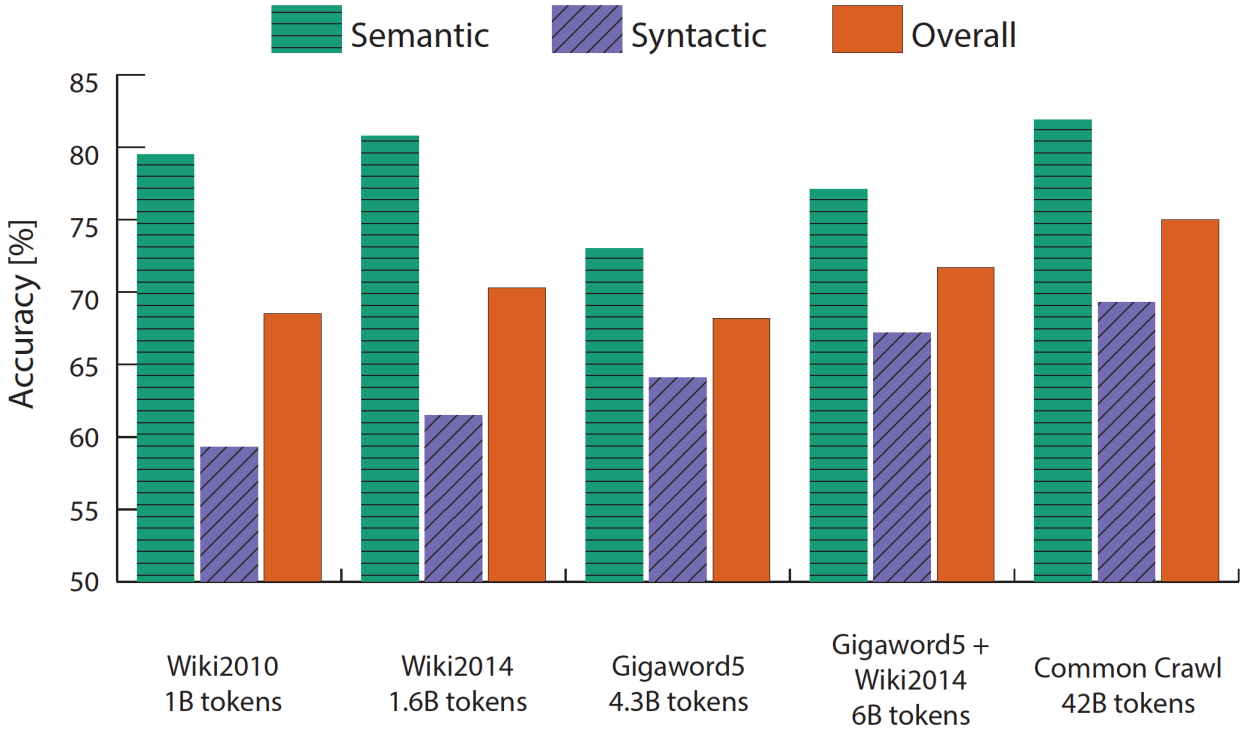
- more training time helps (Glove), more negative samples (word2vec) less so





word2Vec vs GloVe: GloVe: Analogy Task: Performance , [3]

- more data helps, more structured sources targeted at knowledge codification are more helpful than news sources



word2Vec vs GloVe: Intrinsic Eval.: Human Judgement ● , [3]

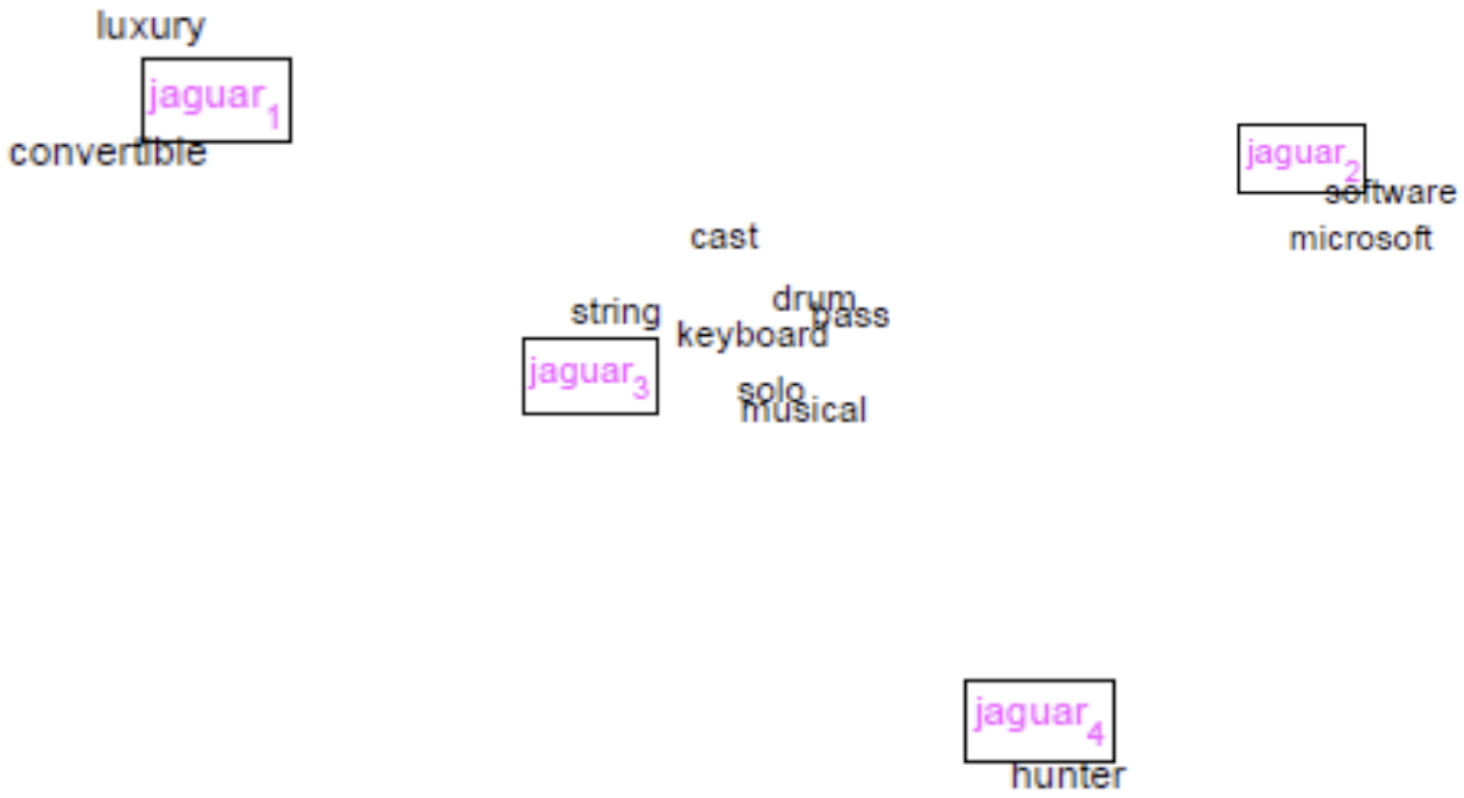
- compute cosine similarity for word vectors and compare to human judgement of similarity on a [0,10] scale via Spearman rank correlations:

Model	Size	WS353	MC	RG	SCWS	RW
SVD	6B	35.3	35.1	42.5	38.3	25.6
SVD-S	6B	56.5	71.5	71.0	53.6	34.7
SVD-L	6B	65.7	72.7	75.1	56.5	37.0
CBOW†	6B	57.2	65.6	68.2	57.0	32.5
SG†	6B	62.8	65.2	69.7	58.1	37.2
GloVe	6B	65.8	72.7	77.8	53.9	38.1
SVD-L	42B	74.0	76.4	74.1	58.3	39.9
GloVe	42B	**75.9**	**83.6**	**82.9**	**59.6**	**47.8**
CBOW*	100B	68.4	79.6	75.4	59.4	45.5

Word 1	Word 2	Human (mean)
tiger	cat	7.35
tiger	tiger	10.00
book	paper	7.46
computer	internet	7.58
plane	car	5.77
professor	doctor	6.62
stock	phone	1.62
stock	CD	1.31
stock	jaguar	0.92

word2Vec vs GloVe: Ambiguity

- different embeddings for different word senses: e.g. find word senses by clustering word-windows, identify word sense in a corpus sentence via cluster model, treat as different word in training



from Huang et al 2012 in [3]

word2Vec vs GloVe: Extrinsic Evaluation: NER Task

, [3]

- F1 score on different NER datasets, dim = 50; Discrete: baseline without word-vectors

Model	Dev	Test	ACE	MUC7
Discrete	91.0	85.4	77.4	73.4
SVD	90.8	85.7	77.3	73.7
SVD-S	91.0	85.5	77.6	74.3
SVD-L	90.5	84.8	73.6	71.5
HPCA	92.6	**88.7**	81.7	80.7
HSMN	90.5	85.7	78.7	74.7
CW	92.2	87.4	81.7	80.2
CBOW	93.1	88.2	82.2	81.1
GloVe	**93.2**	88.3	**82.9**	**82.2**

Network Training

- iid training data-set $\{(\boldsymbol{x}_n, \boldsymbol{y}_n)\}_{n=1}^{N}$, loss function (often (negative) log-likelihood from model) $L(\boldsymbol{y}, \hat{\boldsymbol{y}} = f(\boldsymbol{x}, \boldsymbol{\theta}))$ → total loss $\sum_{n=1}^{N} L(\boldsymbol{y}_n, \hat{\boldsymbol{y}}_n = f(\boldsymbol{x}_n, \boldsymbol{\theta}))$

Algorithm 1 Online Stochastic Gradient Descent Training

Input: Function $f(\mathbf{x}; \theta)$ parameterized with parameters θ.
Input: Training set of inputs $\mathbf{x_1}, \ldots, \mathbf{x_n}$ and outputs $\mathbf{y_1}, \ldots, \mathbf{y_n}$.
Input: Loss function L.
while stopping criteria not met **do**
    Sample a training example $\mathbf{x_i}, \mathbf{y_i}$
    Compute the loss $L(f(\mathbf{x_i}; \theta), \mathbf{y_i})$
    $\hat{\mathbf{g}} \leftarrow$ gradients of $L(f(\mathbf{x_i}; \theta), \mathbf{y_i})$ w.r.t θ
    $\theta \leftarrow \theta + \eta_k \hat{\mathbf{g}}$
return θ

Network Training

Algorithm 2 Minibatch Stochastic Gradient Descent Training

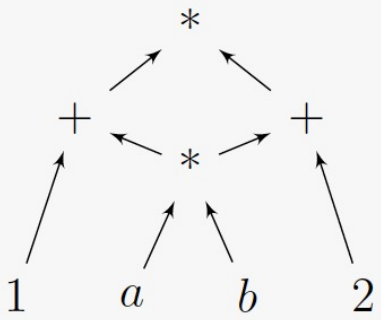
Input: Function $f(\mathbf{x}; \theta)$ parameterized with parameters θ.
Input: Training set of inputs $\mathbf{x_1}, \ldots, \mathbf{x_n}$ and outputs $\mathbf{y_1}, \ldots, \mathbf{y_n}$.
Input: Loss function L.
while stopping criteria not met **do**
    Sample a minibatch of m examples $\{(\mathbf{x_1}, \mathbf{y_1}), \ldots, (\mathbf{x_m}, \mathbf{y_m})\}$
    $\hat{\mathbf{g}} \leftarrow 0$
    for $i = 1$ to m **do**
        Compute the loss $L(f(\mathbf{x_i}; \theta), \mathbf{y_i})$
        $\hat{\mathbf{g}} \leftarrow \hat{\mathbf{g}} +$ gradients of $\frac{1}{m} L(f(\mathbf{x_i}; \theta), \mathbf{y_i})$ w.r.t θ
    $\theta \leftarrow \theta + \eta_k \hat{\mathbf{g}}$
return θ

beyond SGD: SGD + Nesterov Momentum or adaptive learning rate algorithms AdaGrad, AdaDelta, RMSProp, or Adam (see [8])

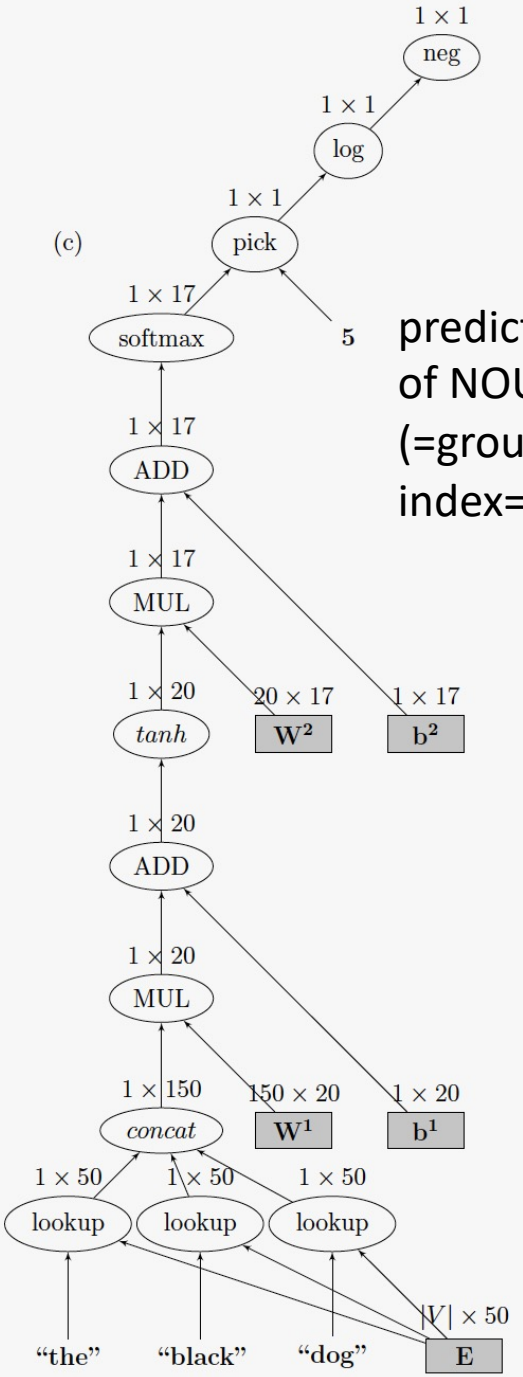
Computation Graph Abstraction

abstract the computations (e.g. forward pass) in a DAG

$(a * b + 1) * (a * b + 2)$:



simple computation



predicted output of NOUN (=ground truth, index=5)

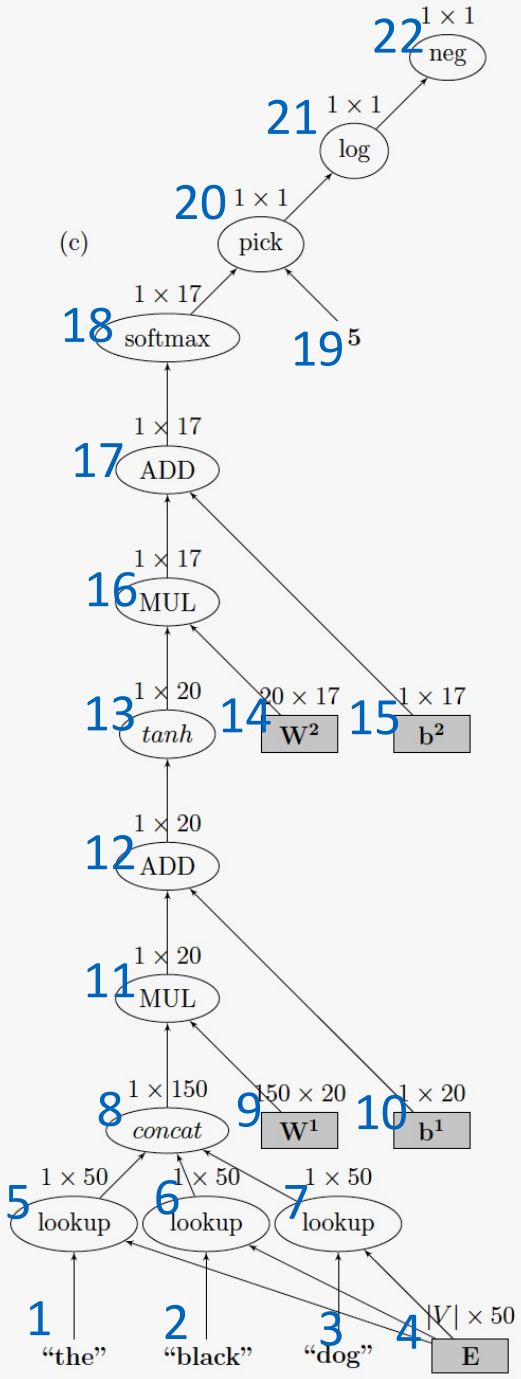
computation graph for a 2 layer FF-NN predicting probability distribution over 17 possible POS for third word of 3 word sequence

Forward Computation

- compute a topological ordering for DAG (V, E) (an ordering such that $i < j$ if $(i, j) \in E$) (possible in linear time)
- f_i : function computed by node i
- $\pi(i)$: children[1] of node i
- $\pi^{-1}(i)$: parents of node i (outputs of $\pi^{-1}(i)$ are the arguments of f_i)
- $v(i)$: output of node i (applying f_i to the output of the $\pi^{-1}(i)$)
- for constants, variable (parameter) nodes and input nodes: f_i is a constant function and $\pi^{-1}(i)$ is empty

Algorithm 3 Computation Graph Forward Pass

1: **for** i = 1 to N **do**
2: Let $a_1, \ldots, a_m = \pi^{-1}(i)$
3: $v(i) \leftarrow f_i(v(a_1), \ldots, v(a_m))$



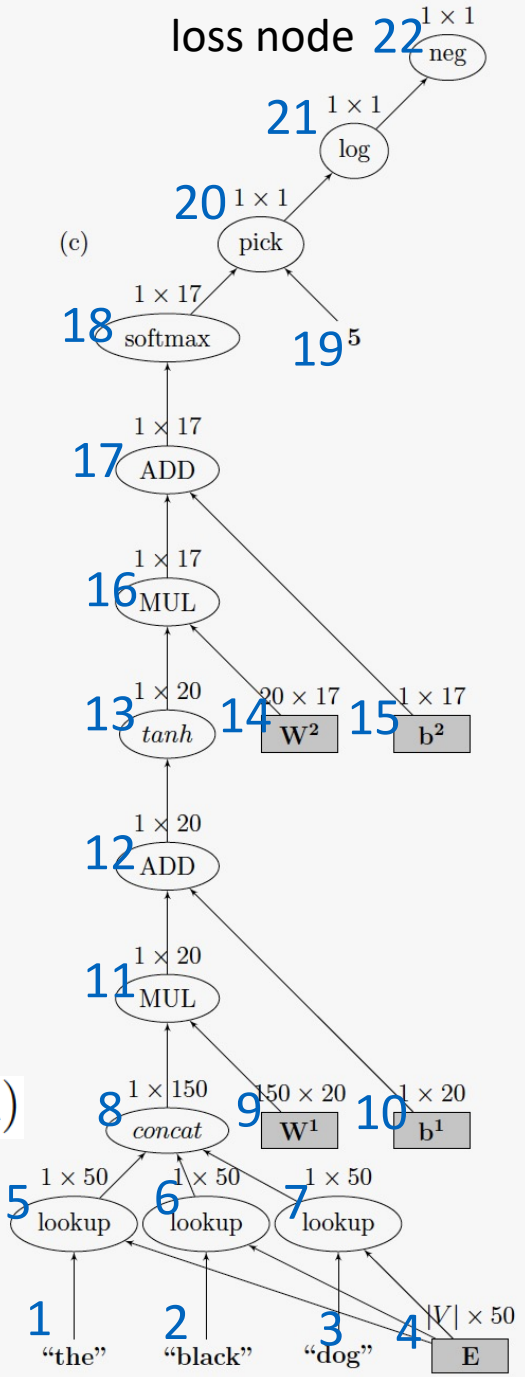
regard that in [1] children and parents are defined in reverse order of the edges; we stick to the usual notion of children and parent: j is child of i and i parent of j if $(i, j) \in E$

Backward Computation

- select the loss node N (usually the last node in the topological ordering)
- $d(i) := \frac{\partial N}{\partial i}$: in the end contains the elements of the gradient w.r.t. the parameter nodes
- $\frac{\partial f_j}{\partial i} = \frac{\partial}{\partial\, i \in \pi^{-1}(j)} f_j(v(\pi^{-1}(j)))$
  where $v(\pi^{-1}(j)) := v(a_1), v(a_2), \ldots, v(a_m)$ are the outputs of $a_1, a_2, \ldots, a_m := \pi^{-1}(j)$
- $\pi(i)$: children of node i

Algorithm 4 Computation Graph Backward Pass (Backpropagation)

1: $d(N) \leftarrow 1$
2: **for** i = N-1 to 1 **do**
3: $\quad d(i) \leftarrow \sum_{j \in \pi(i)} d(j) \cdot \frac{\partial f_j}{\partial i}$



Network Training

Summing it all up:

Algorithm 5 Neural Network Training with Computation Graph Abstraction (using mini-batches of size 1)

```
1: Define network parameters.
2: for iteration = 1 to N do
3:     for Training example x_i, y_i in dataset do
4:         loss_node ← build_computation_graph(x_i, y_i, parameters)
5:         loss_node.forward()
6:         gradients ← loss_node().backward()
7:         parameters ← update_parameters(parameters, gradients)
8: return parameters.
```

In an ideal world: fine tune all meta-parameters (architecture variations (number of layers or neurons, activations, drop-out strategy, other regularization, mini-batch size etc.) as well via systematic evaluation (not via systematic testing ☺)

Optimization Issues

- non-convex loss → training may get stuck in saddle points, local optima → re-run with different random, dimension-dependent initializations of the parameters (Xavier, Gaussian etc.)

- exploding & vanishing gradients (especially in very deep and recurrent architectures)
  - exploding gradients problem (e.g. due to continued multiplication with quantities $|| . || >> 1$):
    solution: gradient clipping:

$$\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|}\hat{\mathbf{g}} \text{ if } \|\hat{\mathbf{g}}\| > threshold$$

  - vanishing gradients problem (e.g. due to continued multiplication with quantities $|| . || << 1$ (e.g. due to saturated neurons)):
    solution: open issue: LSTM or GRU nodes, proper choice of activation function

Optimization Issues

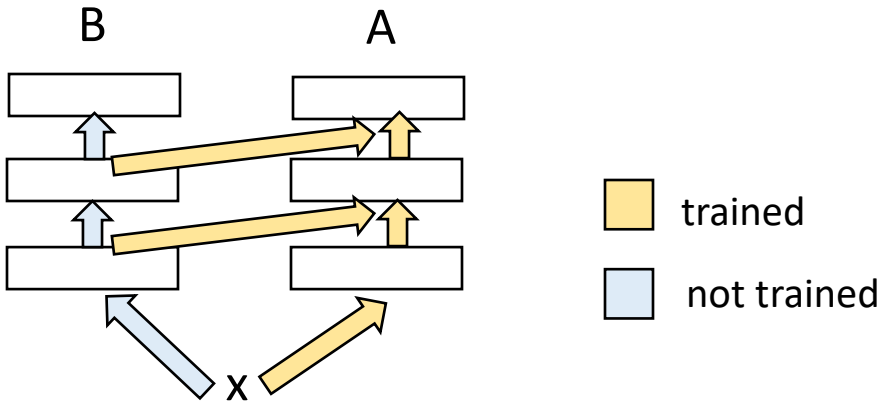
- saturated neurons (especially with $\sigma(x)$ or $\tanh(x)$ for large total inputs x) or dead neurons (relu activation with negative total inputs) → components of gradient for incoming weights of those neurons $|\,.\,| \approx 0$.
  countermeasures:
  - change initialization
  - normalize inputs
  - smaller learning rates
  - saturation: normalize activation:

$$g(\mathbf{h}) = tanh(\mathbf{h}) \rightarrow g(\mathbf{h}) = \frac{tanh(\mathbf{h})}{\|tanh(\mathbf{h})\|}$$

- avoid overfitting → regularization:
  - L2: $\lambda||\boldsymbol{\theta}||$
  - dropout

Model Cascading and Multi-Task Learning

- Cascading / Transfer Learning: from input train auxiliary task B (e.g. POS tagging), feed vector output as input to new network for task A (e.g. chunking);
  - variant: Progressive NN:
    - use same input for both networks
    - feed output of layers of pre-trained network B as additional input to layers of network A
    - train only these additional weights; keep network B fixed



- Multi-Task Learning: provide separate outputs for sufficiently related tasks (e.g. NER, chunking, language modelling), combine into summed loss / alternate btw. different losses → shared intermediate representations.

Structured Output Prediction

- NLP: often structured output desired: e.g. parse trees (e.g. syntactic parsing), tag sequences (e.g. POS tags), or segmentations of sequences (e.g. NER, chunking)

- greedy structured prediction: decompose the structure prediction problem into a sequence of local prediction problems
  - input: typically a sequence
  - output: for every sub-sequence classify current sequence head
  - examples: left to right tagging, greedy transition based parsing
  - disadvantages: error propagation
    → strategies:
    - attempt easier predictions before harder ones
    - focus in training on examples that create likely mistakes

Structured Output Prediction: Search Based

- x: input structure (e.g. sentence); y: output structure (e.g. tag assignment, parse tree); $\mathcal{Y}(x)$: set of valid output structures for x

$$predict(x) = \arg\max_{y \in \mathcal{Y}(x)} score(x, y)$$

example:
1 layer FF-NN

$$score(x, y) = \sum_{p \in parts(x,y)} score(p) = \sum_{p \in parts(x,y)} NN(c(p)) = \sum_{p \in parts(x,y)} (g(c(p)\mathbf{W^1} + \mathbf{b^1}))\mathbf{w}$$

$c(p)$ feature function: encode part p into d^{in} dim. feature vector

- also assumed: inference algorithm: given *score(p)* for all possible p: derive best scoring y

Structured Output Prediction: Search Based

- approach:
  - create computation graphs for computing scores for parts, compute the scores for parts
  - run inference: derive best possible y' (max scoring);
  - using a loss function

$$\max_{y'} score(x, y') - score(x, y)$$

or

$$max(0, m + score(x, y) - \max_{y' \neq y} score(x, y'))$$

(comparing to the ground truth y)
and summing the losses for the parts: backpropagate

Structured Output Prediction: Search Based

- probabilistic (CRF) approach:
  - model:

$p_{(x,y)}$: parts of (x, y)

$$score_{CRF}(x, y) = P(y|x)$$

$$= \frac{\sum_{p_{(x,y)}} \exp(score(p_{(x,y)}))}{\sum_{\tilde{y} \in Y(x)} \sum_{p_{(x,\tilde{y})}} \exp(score(p_{(x,\tilde{y})}))}$$

$$= \frac{\sum_{p_{(x,y)}} \exp(NN(c(p_{(x,y)})))}{\sum_{\tilde{y} \in Y(x)} \sum_{p_{(x,\tilde{y})}} \exp(score(c(p_{(x,\tilde{y})})))}$$

  - loss for a training example (x, y):

$$-\log score_{CRF}(x, y)$$

  - tricky part: denominator (partition function). Usually: exponentially many candidate decompositions into parts: solution use beam search

Convolutional NN for NLP

- example case: sentence-based sentiment classification (+,o,-):
  simple idea: CBOW model using word embeddings + softmaxed FF-NN):
  disadvantage: discarding of word order:

  CBOW(*it was not good, it was actually quite bad*) ==
  CBOW(*it was not bad, it was actually quite good*)

- solution: look for local feature combinations indicative of sentiment (*not good, not bad, quite bad, quite good*): global position in sentence irrelevant but local combination / word order is relevant

- other solution: input concatenation of word embeddings: might not scale well (too many possibilities)

Convolutional NN for NLP

- third solution: use n-gram embeddings; disadvantage: doesn't scale well: sparsity: "quite good" and "very good" are distinct bi-grams: if one is not seen in embedding training it is not possible to deduce anything about the other based on its component words without back-off to single word embeddings

- best solution: CNNs using 1-dim. convolutions ☺

Convolutional NN for NLP

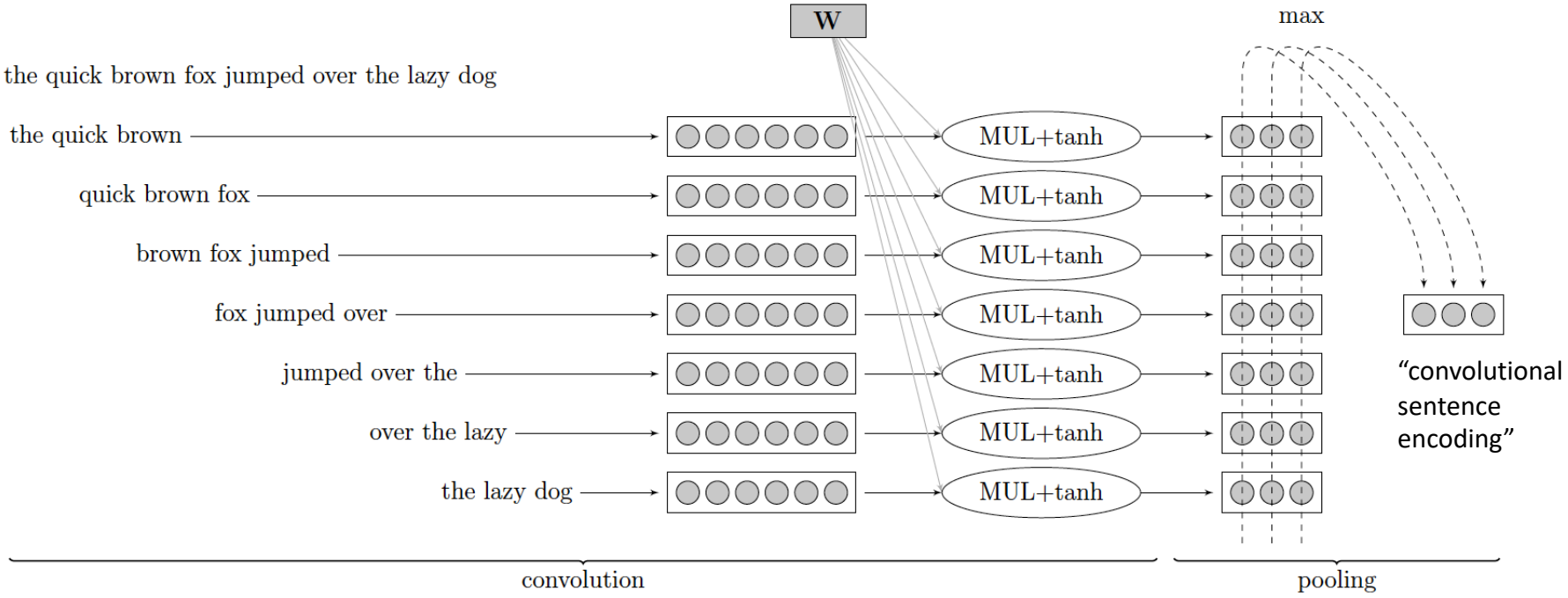
CNNs using 1-dim. convolutions

- sentence = sequence of words $\boldsymbol{w} = w_1, .., w_n$ each with corresponding d^{emb} dim. word embedding $\boldsymbol{v}(w_i)$
- → input $\boldsymbol{x} = [\boldsymbol{v}(w_i); \boldsymbol{v}(w_{i+1}); \ldots ; \boldsymbol{v}(w_{i+k-1})] \in \mathbb{R}^{kd^{emb}}$
- convolution filter: sliding window of size k over the sentence
  - variant A: pad the sentence with $k - 1$ (empty) words to each side (wide convolution) → we get $m = n + k + 1$ window outputs
  - Variant B: do not pad sentence (narrow convolution) → we get $m = n - k - 1$ windows
- → result: m vectors $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_m$ with $\boldsymbol{p}_i = g(\boldsymbol{x_i W} + \boldsymbol{b})$

  where $\boldsymbol{p}_i \in \mathbb{R}^{d^{conv}}$; $\boldsymbol{x} \in \mathbb{R}^{kd^{emb}}$; $\boldsymbol{b} \in \mathbb{R}^{d^{conv}}$; $\boldsymbol{W} \in \mathbb{R}^{kd^{emb} \times d^{conv}}$
- then apply max pooling for each dimension ("channel") of the $\boldsymbol{p}_i$: $c_j = \max_{1<i\leq m} p_{ij}$

Convolutional NN for NLP

- resulting vector ***c***: representation of sentence with a k-gram point of view optimized for some prediction task (via following layers and backprop)



$d^{emb} = 2, \quad k = 3, \quad d^{conv} = 3$

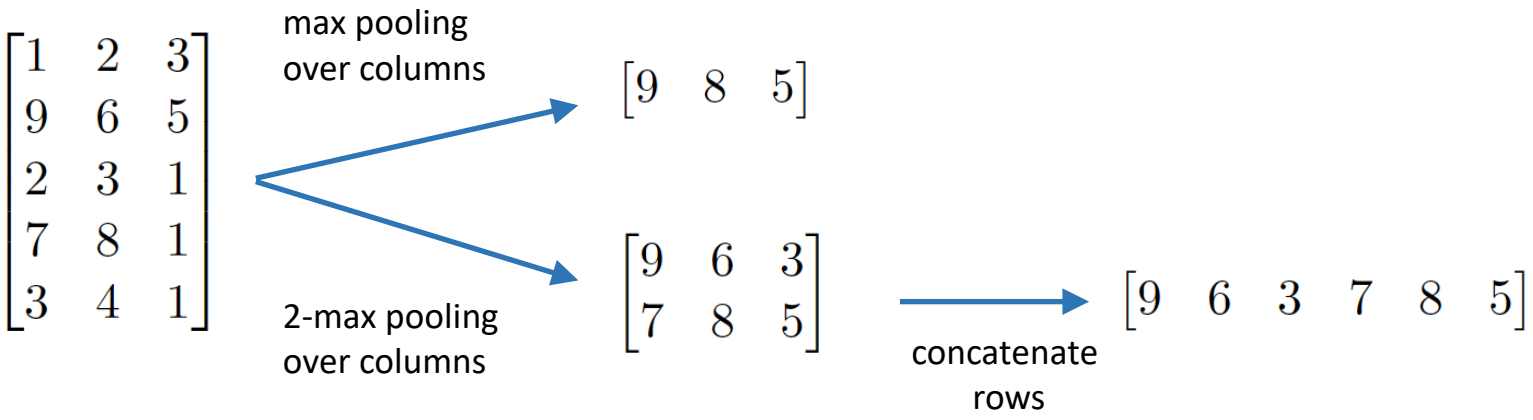
CNN for NLP: Variants of Pooling

- dynamic pooling: instead of pooling over the entire set $\{\boldsymbol{p}_i\}_{i=1}^{m}$, group the $\{\boldsymbol{p}_i\}_{i=1}^{m}$ into $\boldsymbol{l}$ groups and pool separately → $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_l$; reasoning:
  - case document classification into topics: input sequence $\boldsymbol{w}$ = whole document;
    groups = document regions: abstract, introduction, results, discussion, conclusion
  - case relation classification: input sequence $\boldsymbol{w}$ = sentence + two words w_1, w_2 in the sentence;
    groups = regions: before w_1, between w_1, w_2, after w_2

- hierarchical pooling: divide hierarchically and pool for each tree region separately → also include sensitivity to larger structures

CNN for NLP: Variants of Pooling

- k-max pooling: instead of keeping only the maximum, keep the k largest entries.



  - pool the k most active indicators
  - preserves order of the features, but insensitive to specific positions.
  - discern more finely the number of times the feature is highly activated

CNN: Variations

- obvious: use more than one convolutional layer in sequence
  less obvious: use more layers with different window sizes in parallel + concatenate pooled outputs

- use structured input: example: convolution and pooling over syntactic dependency tree:
  - deviate from sequential order (convolution on k grams) → convolution on ancestor path of a node or on node siblings in dependency tree
  - pooling ("max over tree pooling"): over various tree elements

Recurrent NNs (RNNs)

- NLP: typical: sequential input → use RNNs:
  example: language modelling: better perplexity than traditional n-gram based models

$$
\begin{aligned}
RNN(\mathbf{s_0}, \mathbf{x_{1:n}}) &= \mathbf{s_{1:n}}, \ \mathbf{y_{1:n}} \\
\mathbf{s_i} &= R(\mathbf{s_{i-1}}, \mathbf{x_i}) \qquad \mathbf{x_i} \in \mathbb{R}^{d_{in}}, \ \ \mathbf{y_i} \in \mathbb{R}^{d_{out}}, \\
\mathbf{y_i} &= O(\mathbf{s_i}) \qquad \mathbf{s_i} \in \mathbb{R}^{f(d_{out})}
\end{aligned}
$$

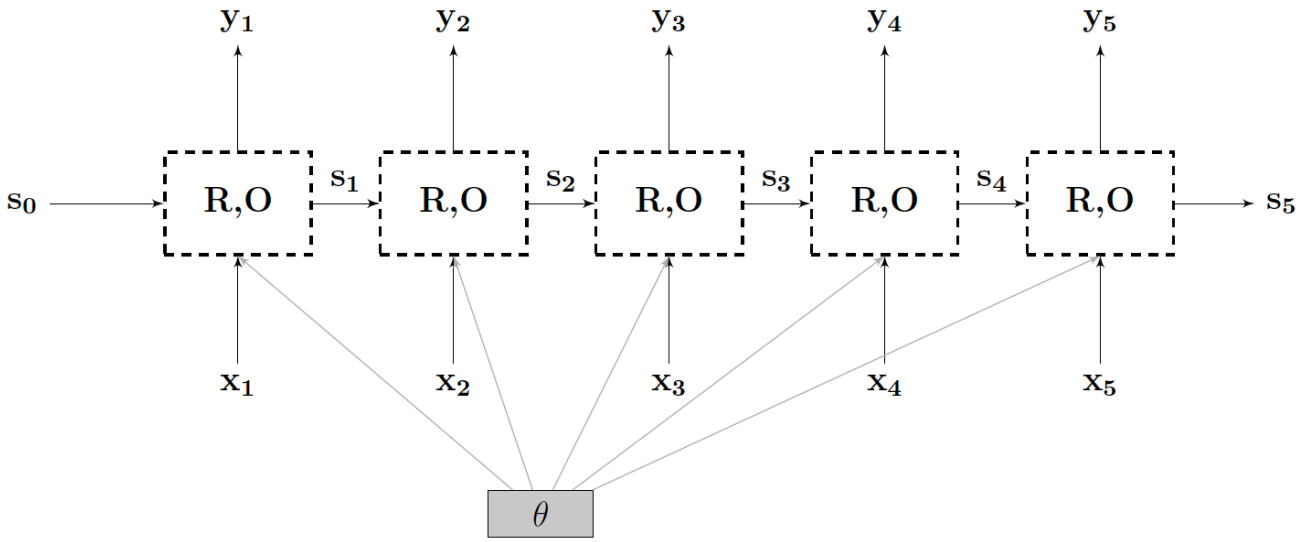
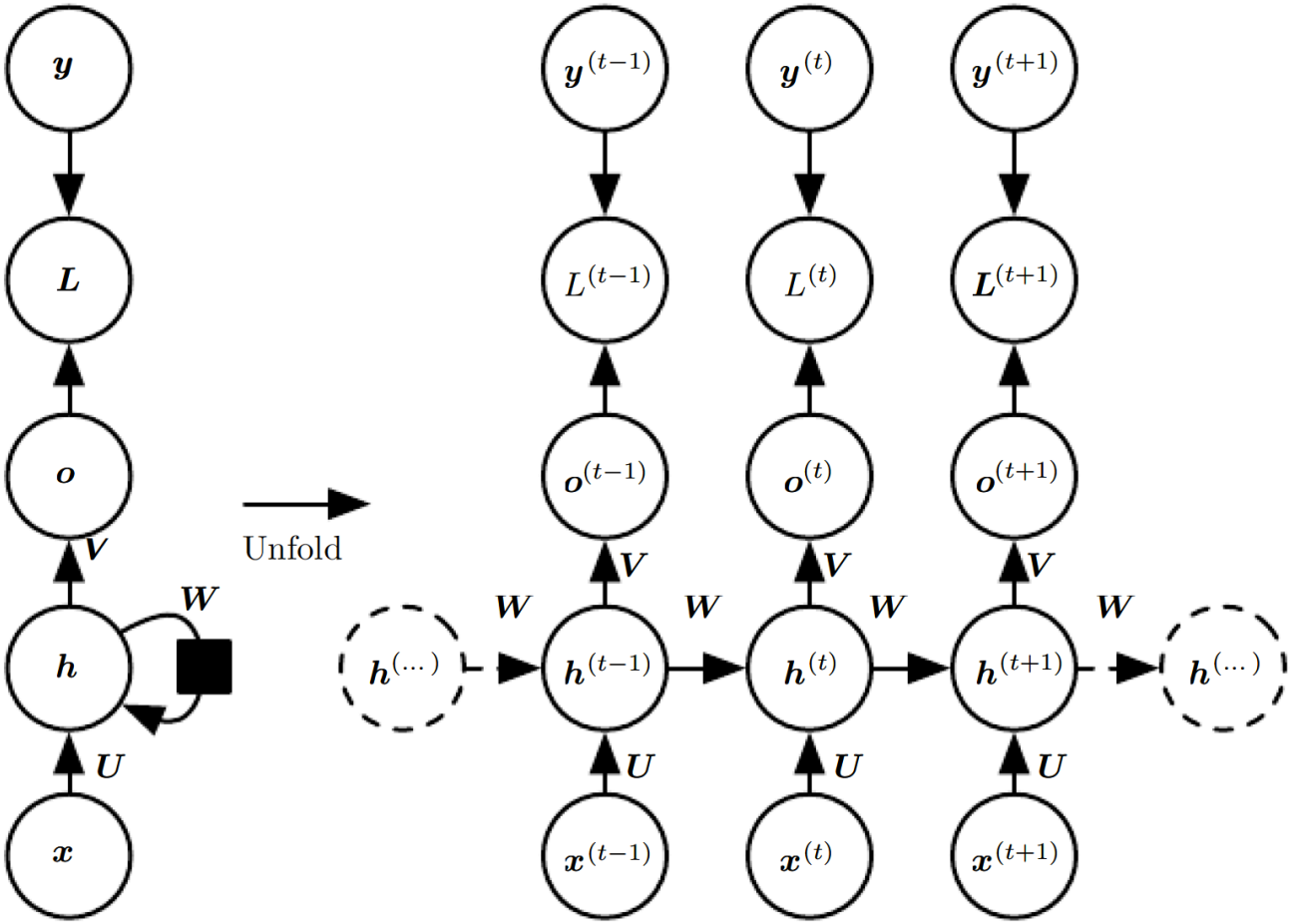


Figure 6: Graphical representation of an RNN (unrolled).

Recurrent NNs (RNNs): Notations

graphical notation in Goodfellow et al. ([4])

$$
\begin{aligned}
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \mathrm{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}
$$

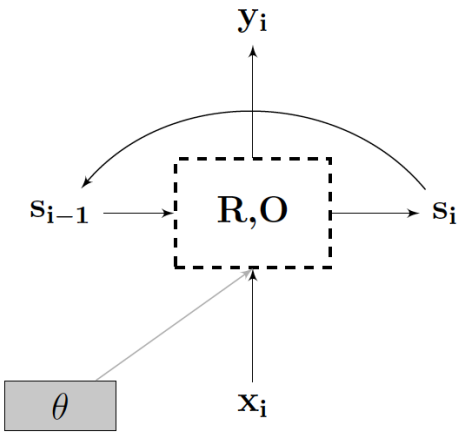


[4]

Recurrent NNs (RNNs): Notations

graphical notation in Goldberg ([1])

$$RNN(\mathbf{s_0}, \mathbf{x_{1:n}}) = \mathbf{s_{1:n}}, \mathbf{y_{1:n}}$$
$$\mathbf{s_i} = R(\mathbf{s_{i-1}}, \mathbf{x_i})$$
$$\mathbf{y_i} = O(\mathbf{s_i})$$



$\mathbf{x_i} \in \mathbb{R}^{d_{in}}$, $\mathbf{y_i} \in \mathbb{R}^{d_{out}}$, $\mathbf{s_i} \in \mathbb{R}^{f(d_{out})}$

Figure 5: Graphical representation of an RNN (recursive).

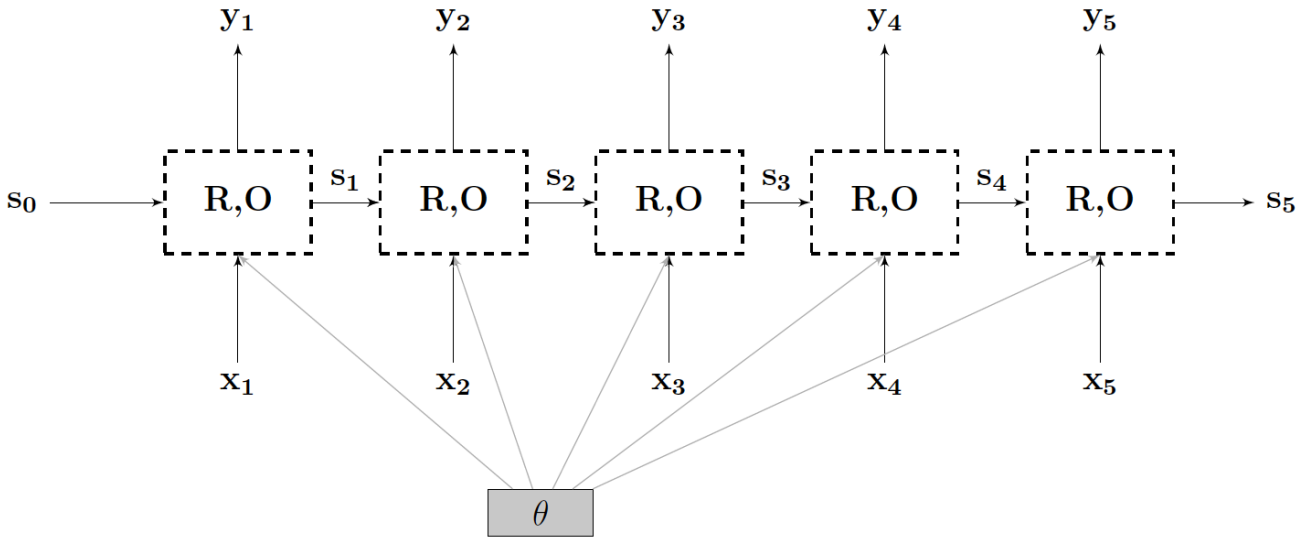


Figure 6: Graphical representation of an RNN (unrolled).

Recurrent NNs (RNNs)

- state vector encodes complete input-sequence over time:

$$
\begin{aligned}
\mathbf{s_4} &= R(\mathbf{s_3}, \mathbf{x_4}) \\
&= R(\overbrace{R(\mathbf{s_2}, \mathbf{x_3})}^{\mathbf{s_3}}, \mathbf{x_4}) \\
&= R(R(\overbrace{R(\mathbf{s_1}, \mathbf{x_2})}^{\mathbf{s_2}}, \mathbf{x_3}), \mathbf{x_4}) \\
&= R(R(R(\overbrace{R(\mathbf{s_0}, \mathbf{x_1})}^{\mathbf{s_1}}, \mathbf{x_2}), \mathbf{x_3}), \mathbf{x_4})
\end{aligned}
$$

→ s_n and y_n may be viewed as encoding the complete sequence

- but: later inputs may have greater influence than earlier ones: need to control forgetting (e.g. via Attention or by using LSTMs)

RNN Training

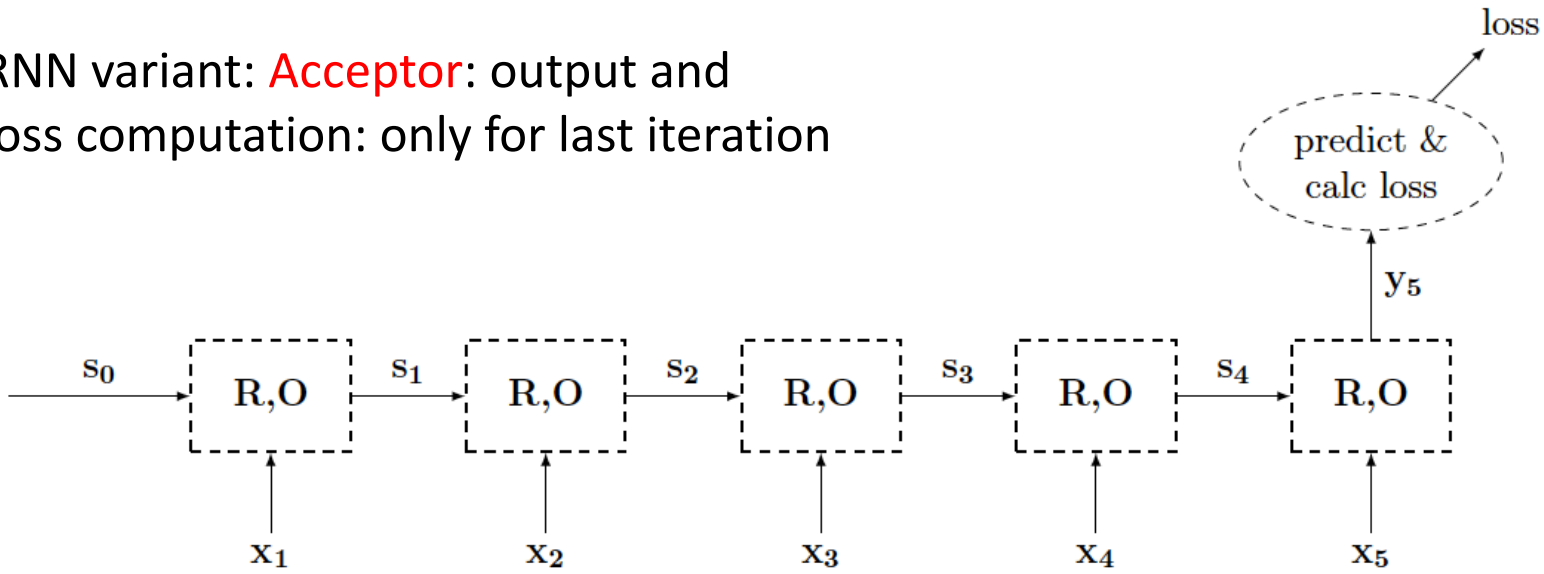
- unrolled computation graph: just as for any other (FF) NN → use backpropagation over unrolled computation graph (backpropagation through time (BPTT))

- vanishing gradient → variant (especially when using non-LSTM or GRU nodes): unroll only a limited number of iterations

- we will later look into vanishing gradients and why gated nodes (GRUs and LSTMs) are a good contribution towards solving this issue!

RNN Training: Acceptors and Encoders

- RNN variant: Acceptor: output and loss computation: only for last iteration

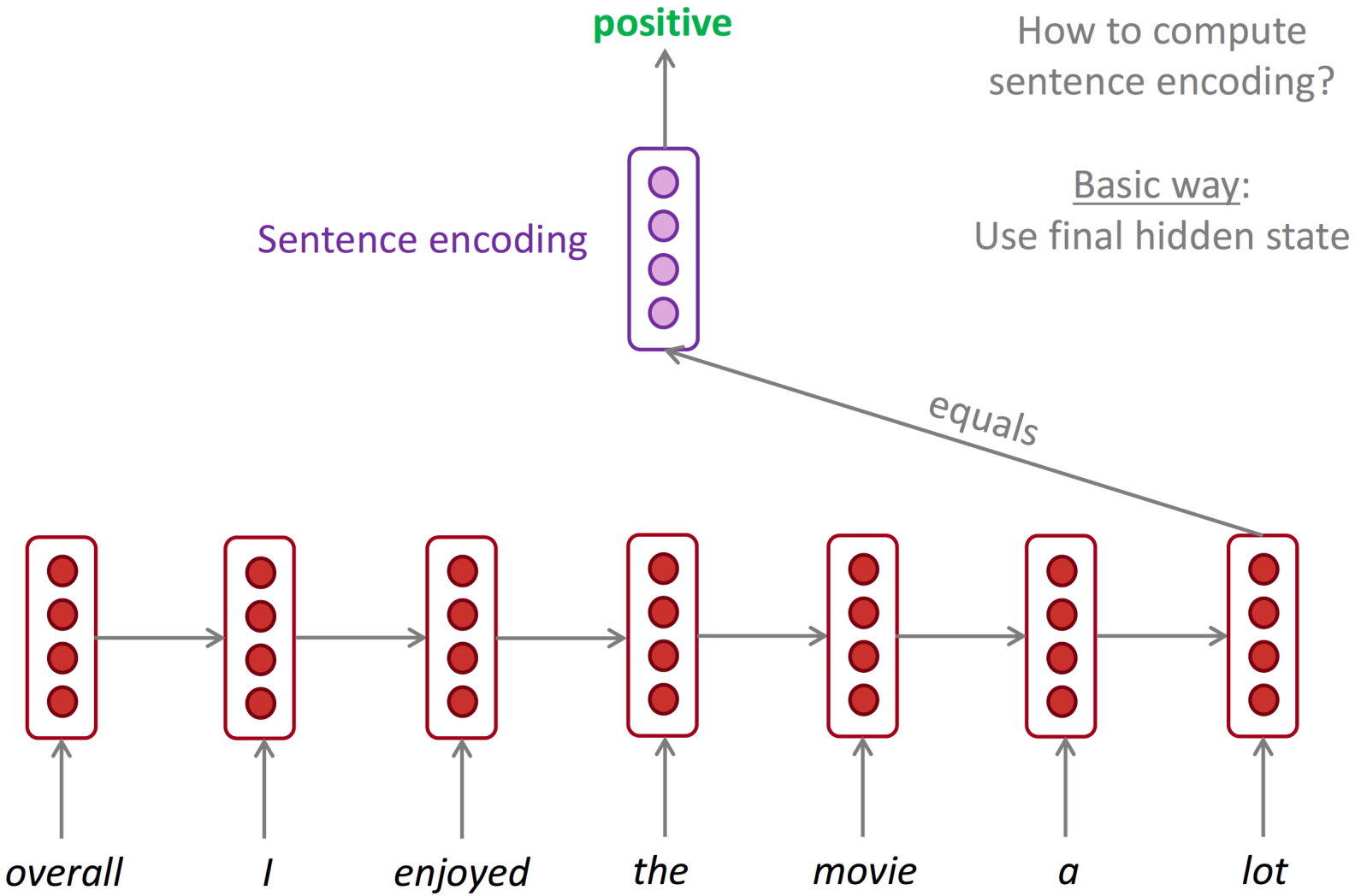


  - examples:
    - read characters of a word before predicting POS
    - read sentence before classifying its total sentiment
    - read word sequence and classify as parse element (e.g. noun-phrase)

- Enocder: just as Acceptor but make final prediction using output $\boldsymbol{y}_n$ AND other features from sequence;
  - example: auto summarization: first read all sentences, encoding them in $\boldsymbol{y}_n$, then decide on sentences to include in summarization

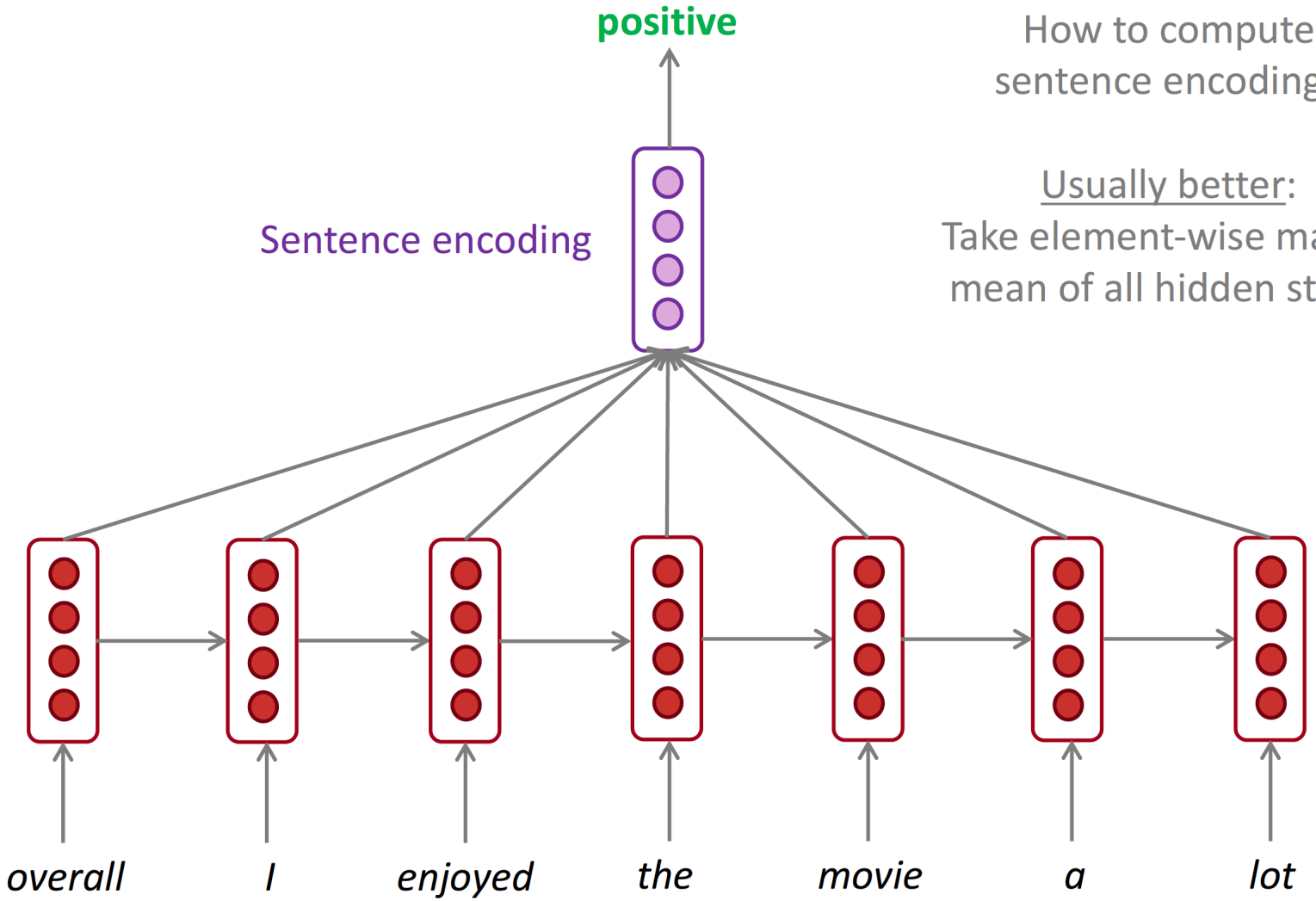
RNN Training: Acceptors: Example

e.g. sentiment classification



RNN Training: (…): Example

e.g. sentiment classification



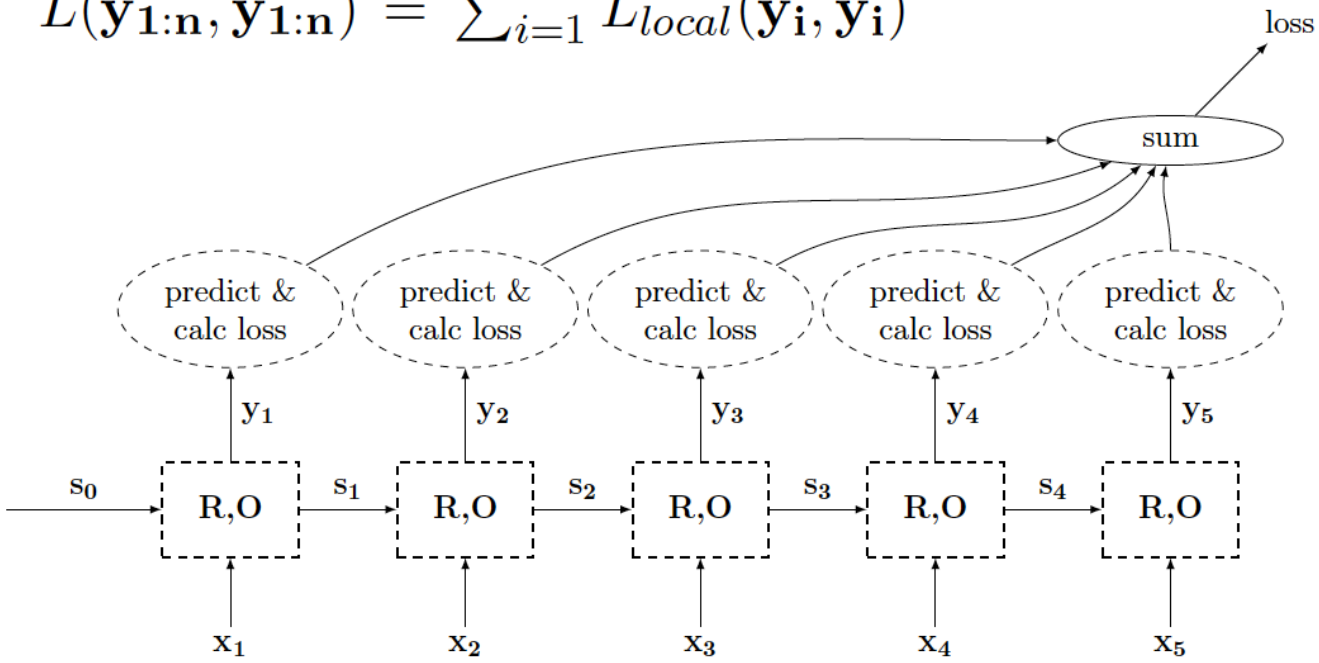
How to compute sentence encoding?

Usually better:
Take element-wise max or mean of all hidden states

RNN Training: Transducers

- output for every iteration; loss for unrolled sequence:

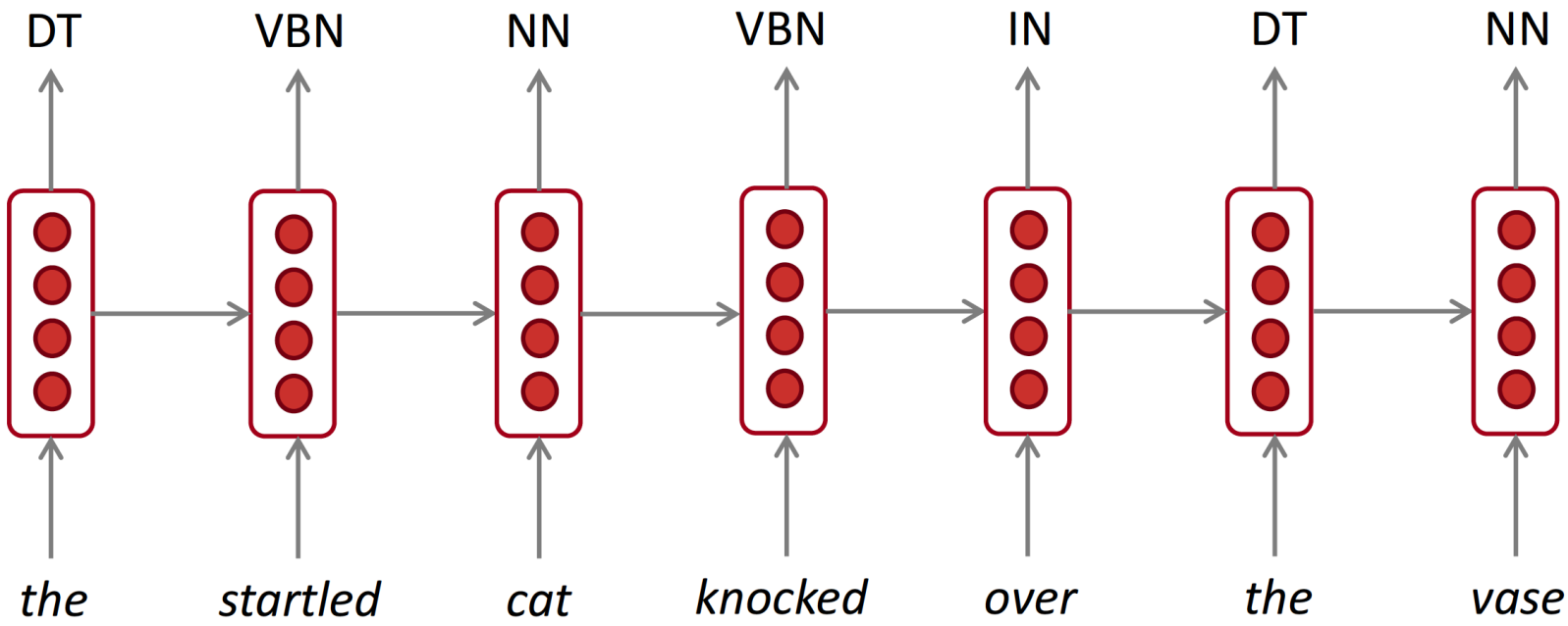
$$L(\hat{\mathbf{y}}_{1:n}, \mathbf{y}_{1:n}) = \sum_{i=1}^{n} L_{local}(\hat{\mathbf{y}}_i, \mathbf{y}_i)$$



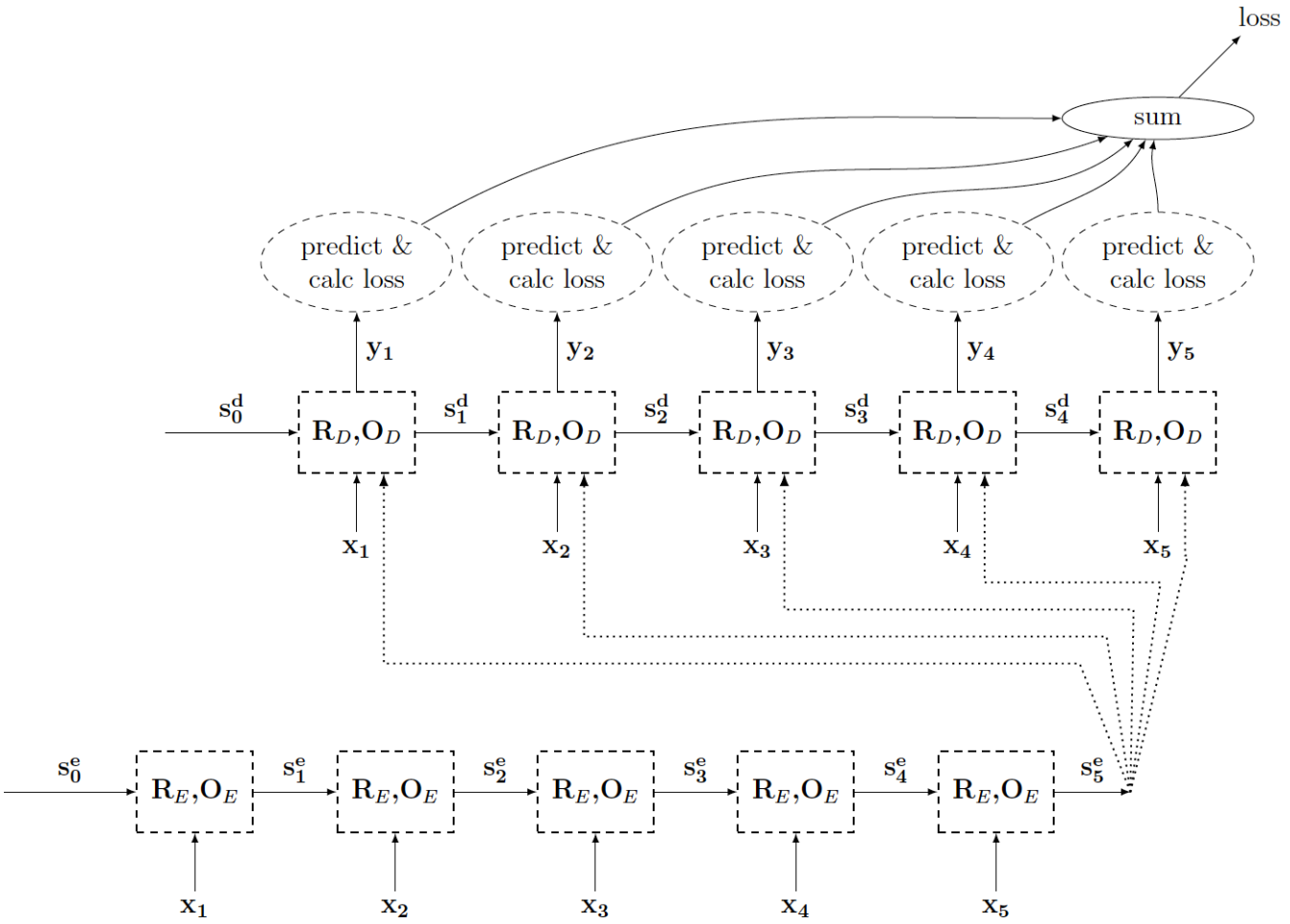
- typical application: sequence tagging (e.g. POS tagging or CCG supertagging) or language modelling
- power to condition output beyond first order Markov: demonstrated by generative character-level text generation models

RNN Training: Transducers: Example

e.g. part-of-speech tagging, named entity recognition



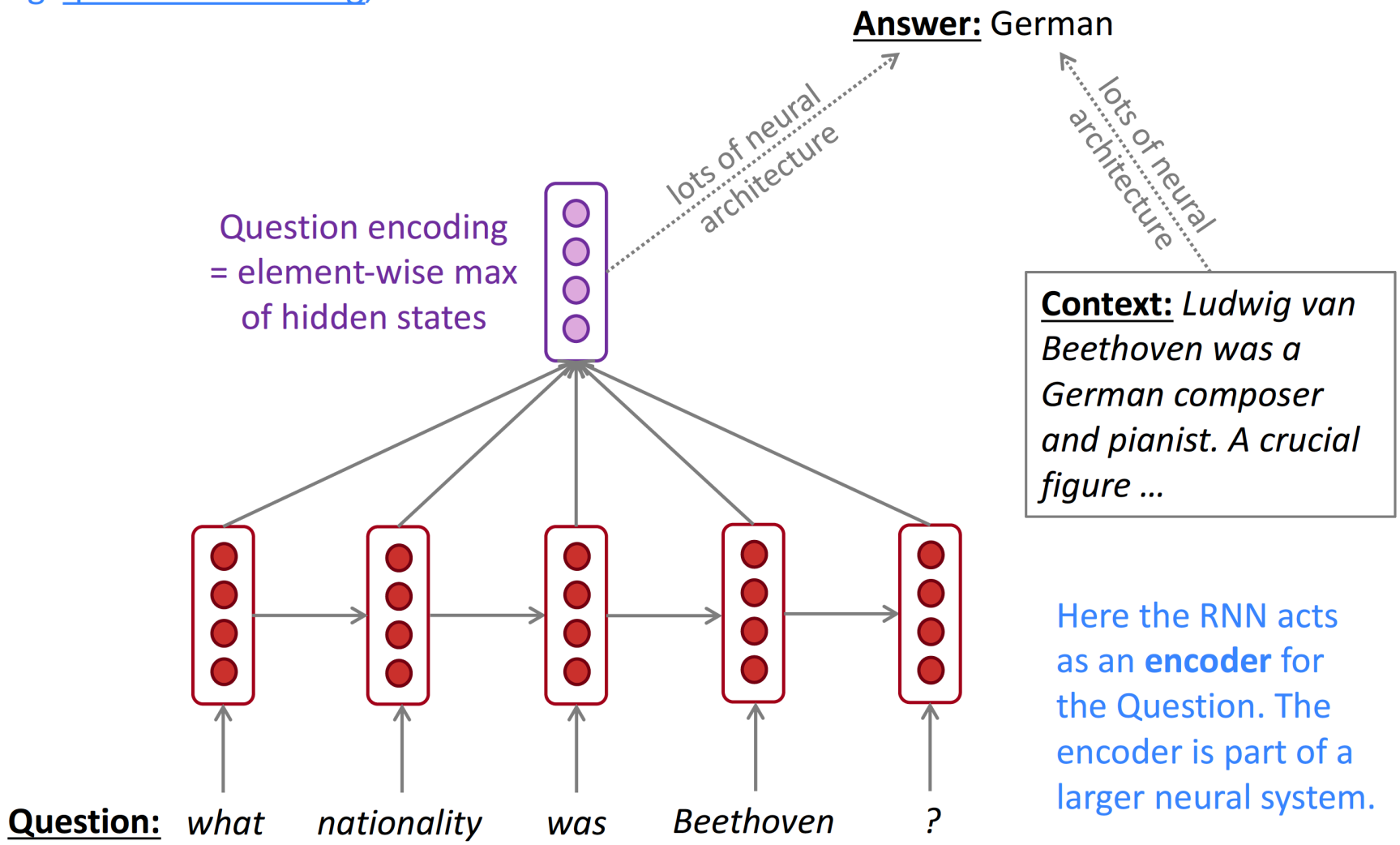
RNN Training: Encoder – Decoder (seq2seq)



- examples:
  - machine translation (maybe advisable to input source sentence (also) in reverse order)
  - sequence tagging, using encoder's $\boldsymbol{y}_n$ as initial state of transducer RNN

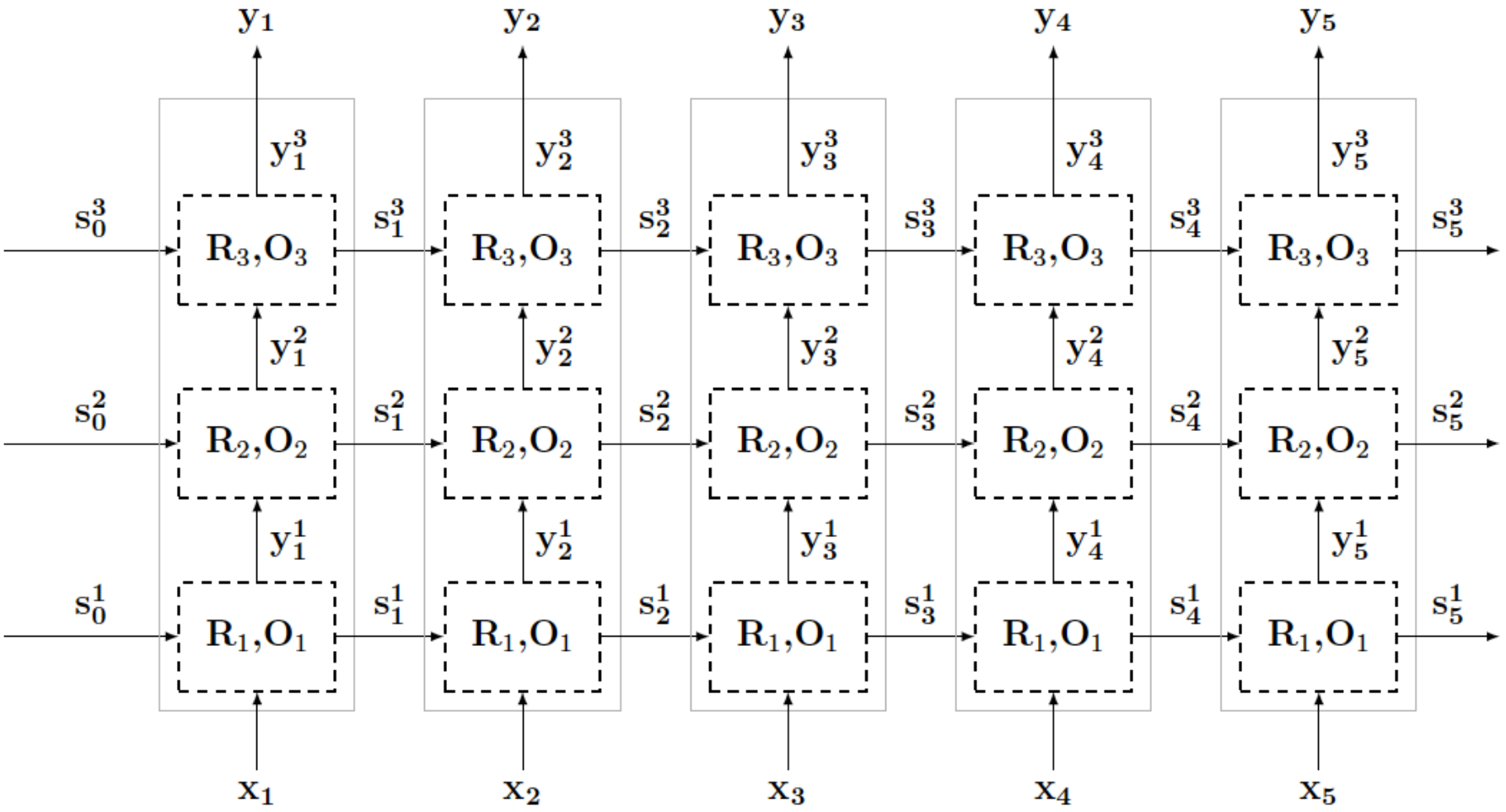
RNN Training: Encoder + Stuff ☺: Example

e.g. question answering, machine translation



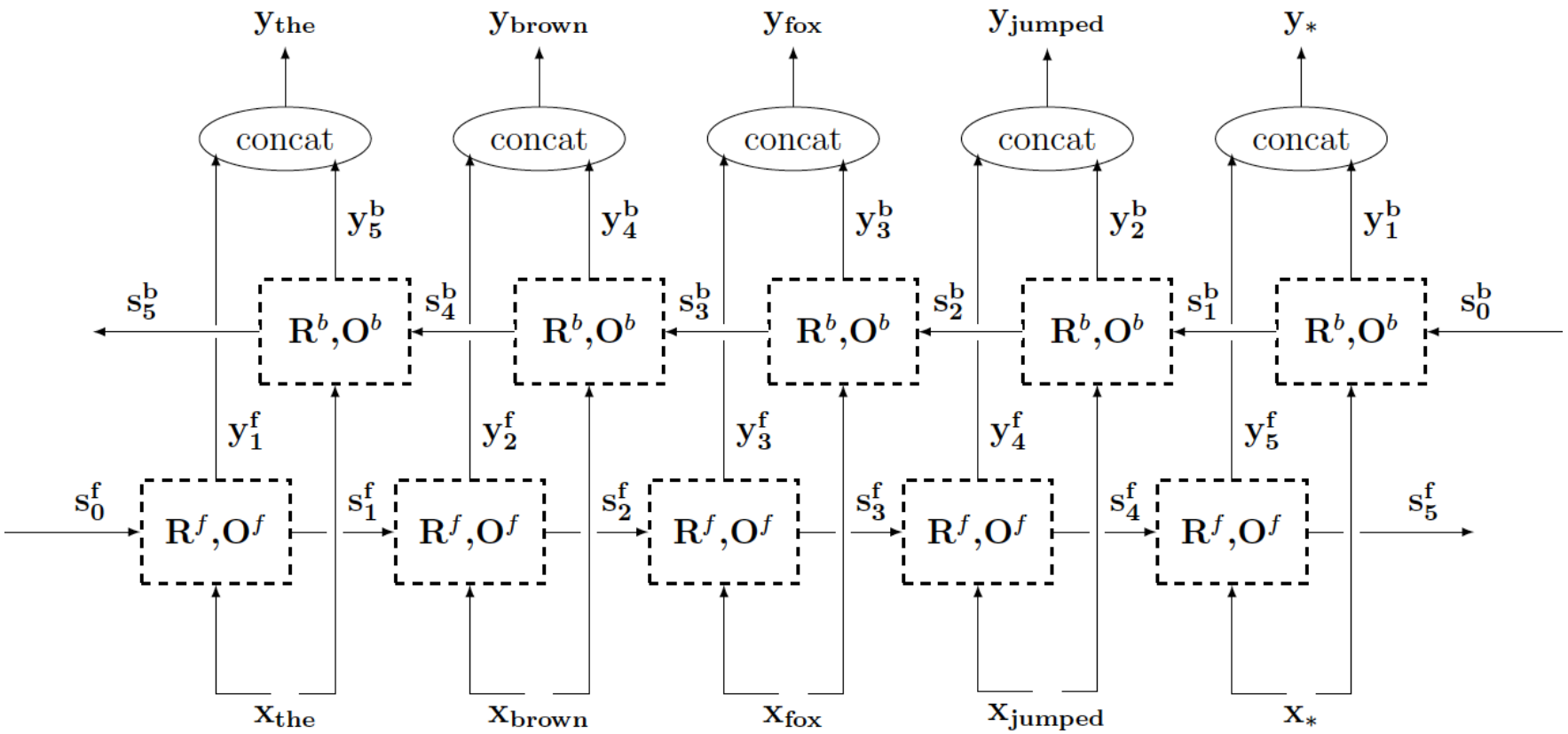
Here the RNN acts as an **encoder** for the Question. The encoder is part of a larger neural system.

Multi-Layer RNNs



Bi-RNNs

two component RNNs: one for backward and one for forward direction: look into past and into future:



RNNs for Stacks

- idea: for applications using stacks (e.g. transition-based dependency parsing), not just look at top k elements of stack but encode whole stack with RNN
- for efficiency: encode stack + complete operation history into immutable stack data structure (e.g. a tree)

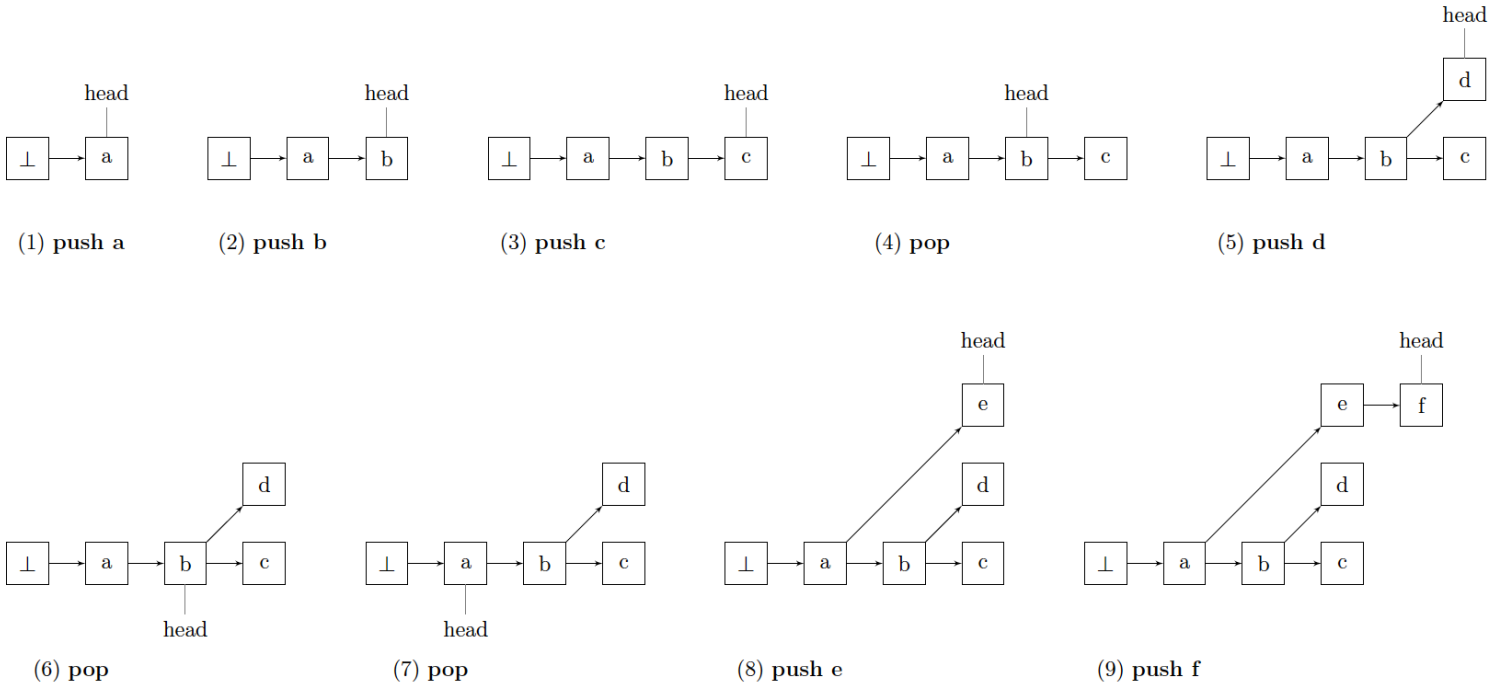


Figure 12: An immutable stack construction for the sequence of operations *push a; push b; push c; pop; push d; pop; pop; push e; push f.*

RNNs for Stacks

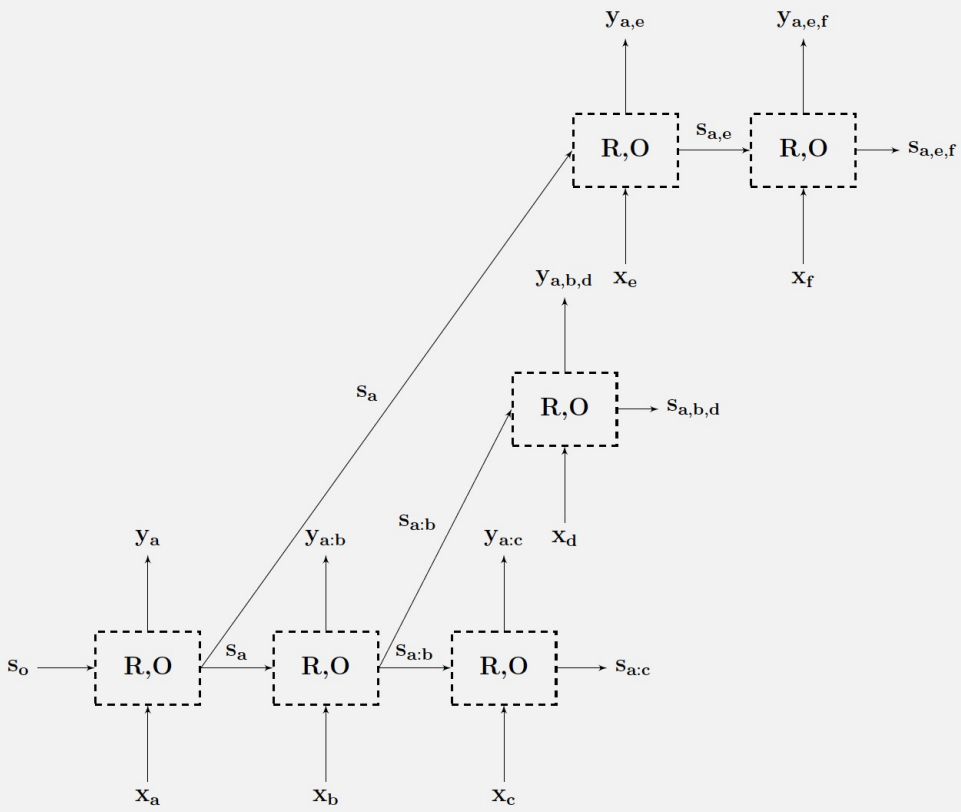
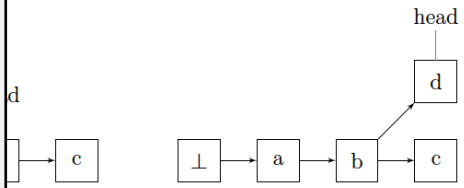
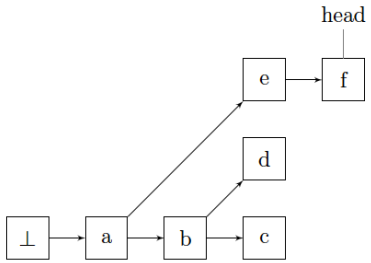


Figure 13: The stack-RNN corresponding to the final state in Figure 12.

poral step;
pping stack content and
ion-based dependency



(5) **push d**



(9) **push f**

Figure 12: An immutable stack construction for the sequence of operations *push a; push b; push c; pop; push d; pop; pop; push e; push f.*

LSTMs

- straightforward RNNs: BPTT → vanishing gradients over time → hard for RNN to learn long range dependencies

- → control memory with the help of gates: decide how much of the new input should be written to the memory cell, and how much of the current content of the memory cell should be forgotten

LSTMs

- straightforward RNNs: BPTT → vanishing gradients over time → hard for RNN to learn long range dependencies

- → control memory with the help of gates: decide how much of the new input should be written to the memory cell, and how much of the current content of the memory cell should be forgotten

$$
\begin{aligned}
\mathbf{s_j} = R_{LSTM}(\mathbf{s_{j-1}}, \mathbf{x_j}) &= [\mathbf{c_j}; \mathbf{h_j}] \\
\mathbf{c_j} &= \mathbf{c_{j-1}} \odot \mathbf{f} + \mathbf{g} \odot \mathbf{i} \\
\mathbf{h_j} &= \tanh(\mathbf{c_j}) \odot \mathbf{o} \\
\mathbf{i} &= \sigma(\mathbf{x_j}\mathbf{W^{xi}} + \mathbf{h_{j-1}}\mathbf{W^{hi}}) \\
\mathbf{f} &= \sigma(\mathbf{x_j}\mathbf{W^{xf}} + \mathbf{h_{j-1}}\mathbf{W^{hf}}) \\
\mathbf{o} &= \sigma(\mathbf{x_j}\mathbf{W^{xo}} + \mathbf{h_{j-1}}\mathbf{W^{ho}}) \\
\mathbf{g} &= \tanh(\mathbf{x_j}\mathbf{W^{xg}} + \mathbf{h_{j-1}}\mathbf{W^{hg}})
\end{aligned}
$$

$$
\mathbf{y_j} = O_{LSTM}(\mathbf{s_j}) = \mathbf{h_j}
$$

$$
\mathbf{s_j} \in \mathbb{R}^{2 \cdot d_h}, \ \mathbf{x_i} \in \mathbb{R}^{d_x}, \ \mathbf{c_j}, \mathbf{h_j}, \mathbf{i}, \mathbf{f}, \mathbf{o}, \mathbf{g} \in \mathbb{R}^{d_h},
$$

$$
\mathbf{W^{xo}} \in \mathbb{R}^{d_x \times d_h}, \ \mathbf{W^{ho}} \in \mathbb{R}^{d_h \times d_h}
$$

LSTMs

- straightforward RNNs: BPTT → vanishing gradients over time → hard for RNN to learn long range dependencies

- → control memory with the help of gates: decide how much of the new input should be written to the memory cell, and how much of the current content of the memory cell should be forg...

memory component

output component

together form state s_j

$$\mathbf{s_j} = R_{LSTM}(\mathbf{s_{j-1}}, \mathbf{x_j}) = [\mathbf{c_j}; \mathbf{h_j}]$$

$$\mathbf{c_j} = \mathbf{c_{j-1}} \odot \mathbf{f} + \mathbf{g} \odot \mathbf{i}$$

$$\mathbf{h_j} = \tanh(\mathbf{c_j}) \odot \mathbf{o}$$

$$\mathbf{i} = \sigma(\mathbf{x_j}\mathbf{W^{xi}} + \mathbf{h_{j-1}}\mathbf{W^{hi}})$$

$$\mathbf{f} = \sigma(\mathbf{x_j}\mathbf{W^{xf}} + \mathbf{h_{j-1}}\mathbf{W^{hf}})$$

$$\mathbf{o} = \sigma(\mathbf{x_j}\mathbf{W^{xo}} + \mathbf{h_{j-1}}\mathbf{W^{ho}})$$

$$\mathbf{g} = \tanh(\mathbf{x_j}\mathbf{W^{xg}} + \mathbf{h_{j-1}}\mathbf{W^{hg}})$$

$$\mathbf{y_j} = O_{LSTM}(\mathbf{s_j}) = \mathbf{h_j}$$

$$\mathbf{s_j} \in \mathbb{R}^{2 \cdot d_h}, \ \mathbf{x_i} \in \mathbb{R}^{d_x}, \ \mathbf{c_j}, \mathbf{h_j}, \mathbf{i}, \mathbf{f}, \mathbf{o}, \mathbf{g} \in \mathbb{R}^{d_h},$$

$$\mathbf{W^{xo}} \in \mathbb{R}^{d_x \times d_h}, \ \mathbf{W^{ho}} \in \mathbb{R}^{d_h \times d_h}$$

LSTMs

- straightforward RNNs: BPTT → vanishing gradients over time → hard for RNN to learn long range dependencies

- → control memory with the help of gates: decide how much of the new input should be written to the memory cell, and how much of the current content of the memory cell should be forg...

memory component

output component

$$\mathbf{s_j} = R_{LSTM}(\mathbf{s_{j-1}}, \mathbf{x_j}) = [\mathbf{c_j}; \mathbf{h_j}]$$

together form state s_j

$$\mathbf{c_j} = \mathbf{c_{j-1}} \odot \mathbf{f} + \mathbf{g} \odot \mathbf{i}$$

$$\mathbf{h_j} = \tanh(\mathbf{c_j}) \odot \mathbf{o}$$

$$\mathbf{i} = \sigma(\mathbf{x_j}\mathbf{W^{xi}} + \mathbf{h_{j-1}}\mathbf{W^{hi}})$$ input gate

$$\mathbf{f} = \sigma(\mathbf{x_j}\mathbf{W^{xf}} + \mathbf{h_{j-1}}\mathbf{W^{hf}})$$ forget gate

$$\mathbf{o} = \sigma(\mathbf{x_j}\mathbf{W^{xo}} + \mathbf{h_{j-1}}\mathbf{W^{ho}})$$ output gate

$$\mathbf{g} = \tanh(\mathbf{x_j}\mathbf{W^{xg}} + \mathbf{h_{j-1}}\mathbf{W^{hg}})$$

$$\mathbf{y_j} = O_{LSTM}(\mathbf{s_j}) = \mathbf{h_j}$$

$$\mathbf{s_j} \in \mathbb{R}^{2 \cdot d_h}, \ \mathbf{x_i} \in \mathbb{R}^{d_x}, \ \mathbf{c_j}, \mathbf{h_j}, \mathbf{i}, \mathbf{f}, \mathbf{o}, \mathbf{g} \in \mathbb{R}^{d_h},$$

$$\mathbf{W^{xo}} \in \mathbb{R}^{d_x \times d_h}, \ \mathbf{W^{ho}} \in \mathbb{R}^{d_h \times d_h}$$

LSTMs

- straightforward RNNs: BPTT → vanishing gradients over time → hard for RNN to learn long range dependencies

- → control memory with the help of gates: decide how much of the new input should be written to the memory cell, and how much of the current content of the memory cell should be forg...

memory component

output component

$$\mathbf{s_j} = R_{LSTM}(\mathbf{s_{j-1}}, \mathbf{x_j}) = [\mathbf{c_j}; \mathbf{h_j}]$$

together form state s_j

$$\mathbf{c_j} = \mathbf{c_{j-1}} \odot \mathbf{f} + \mathbf{g} \odot \mathbf{i}$$

$$\mathbf{h_j} = \tanh(\mathbf{c_j}) \odot \mathbf{o}$$

$$\mathbf{i} = \sigma(\mathbf{x_j}\mathbf{W^{xi}} + \mathbf{h_{j-1}}\mathbf{W^{hi}}) \quad \text{input gate}$$

$$\mathbf{f} = \sigma(\mathbf{x_j}\mathbf{W^{xf}} + \mathbf{h_{j-1}}\mathbf{W^{hf}}) \quad \text{forget gate}$$

$$\mathbf{o} = \sigma(\mathbf{x_j}\mathbf{W^{xo}} + \mathbf{h_{j-1}}\mathbf{W^{ho}}) \quad \text{output gate}$$

$$\mathbf{g} = \tanh(\mathbf{x_j}\mathbf{W^{xg}} + \mathbf{h_{j-1}}\mathbf{W^{hg}}) \quad \text{update candidate}$$

$$\mathbf{y_j} = O_{LSTM}(\mathbf{s_j}) = \mathbf{h_j}$$

$$\mathbf{s_j} \in \mathbb{R}^{2 \cdot d_h}, \ \mathbf{x_i} \in \mathbb{R}^{d_x}, \ \mathbf{c_j}, \mathbf{h_j}, \mathbf{i}, \mathbf{f}, \mathbf{o}, \mathbf{g} \in \mathbb{R}^{d_h},$$

$$\mathbf{W^{xo}} \in \mathbb{R}^{d_x \times d_h}, \ \mathbf{W^{ho}} \in \mathbb{R}^{d_h \times d_h}$$

LSTMs

- straightforward RNNs: BPTT → vanishing gradients over time → hard for RNN to learn long range dependencies

- → control memory with the help of gates: decide how much of the new input should be written to the memory cell, and how much of the current content of the memory cell should be forg...

memory component

output component

$$\mathbf{s_j} = R_{LSTM}(\mathbf{s_{j-1}}, \mathbf{x_j}) = [\mathbf{c_j}; \mathbf{h_j}]$$

together form state s_j

$$\mathbf{c_j} = \mathbf{c_{j-1}} \odot \mathbf{f} + \mathbf{g} \odot \mathbf{i}$$

$$\mathbf{h_j} = \tanh(\mathbf{c_j}) \odot \mathbf{o}$$

$$\mathbf{i} = \sigma(\mathbf{x_j}\mathbf{W^{xi}} + \mathbf{h_{j-1}}\mathbf{W^{hi}}) \text{ input gate}$$

$$\mathbf{f} = \sigma(\mathbf{x_j}\mathbf{W^{xf}} + \mathbf{h_{j-1}}\mathbf{W^{hf}}) \text{ forget gate}$$

$$\mathbf{o} = \sigma(\mathbf{x_j}\mathbf{W^{xo}} + \mathbf{h_{j-1}}\mathbf{W^{ho}}) \text{ output gate}$$

$$\mathbf{g} = \tanh(\mathbf{x_j}\mathbf{W^{xg}} + \mathbf{h_{j-1}}\mathbf{W^{hg}}) \text{ update candidate}$$

component-wise (Hadamard) product:

$$\begin{pmatrix}4\\2\\6\end{pmatrix} \odot \begin{pmatrix}1\\0\\2\end{pmatrix} = \begin{pmatrix}4\\0\\12\end{pmatrix}$$

$$\mathbf{y_j} = O_{LSTM}(\mathbf{s_j}) = \mathbf{h_j}$$

$$\mathbf{s_j} \in \mathbb{R}^{2 \cdot d_h}, \ \mathbf{x_i} \in \mathbb{R}^{d_x}, \ \mathbf{c_j}, \mathbf{h_j}, \mathbf{i}, \mathbf{f}, \mathbf{o}, \mathbf{g} \in \mathbb{R}^{d_h},$$

$$\mathbf{W^{xo}} \in \mathbb{R}^{d_x \times d_h}, \ \mathbf{W^{ho}} \in \mathbb{R}^{d_h \times d_h}$$

LSTMs

- straightforward RNNs: BPTT → vanishing gradients over time → hard for RNN to learn long range dependencies

- → control memory with the help of gates: decide how much of the new input should be written to the memory cell, and how much of the current content of the memory cell should be forg...

memory component

output component

$$\mathbf{s_j} = R_{LSTM}(\mathbf{s_{j-1}}, \mathbf{x_j}) = [\mathbf{c_j}; \mathbf{h_j}]$$

together form state $\boldsymbol{s_j}$

$$\mathbf{c_j} = \mathbf{c_{j-1}} \odot \mathbf{f} + \mathbf{g} \odot \mathbf{i}$$

$$\mathbf{h_j} = \tanh(\mathbf{c_j}) \odot \mathbf{o}$$

$$\mathbf{i} = \sigma(\mathbf{x_j}\mathbf{W^{xi}} + \mathbf{h_{j-1}}\mathbf{W^{hi}}) \quad \text{input gate}$$

$$\mathbf{f} = \sigma(\mathbf{x_j}\mathbf{W^{xf}} + \mathbf{h_{j-1}}\mathbf{W^{hf}}) \quad \text{forget gate}$$

$$\mathbf{o} = \sigma(\mathbf{x_j}\mathbf{W^{xo}} + \mathbf{h_{j-1}}\mathbf{W^{ho}}) \quad \text{output gate}$$

$$\mathbf{g} = \tanh(\mathbf{x_j}\mathbf{W^{xg}} + \mathbf{h_{j-1}}\mathbf{W^{hg}}) \quad \text{update candidate}$$

component-wise (Hadamard) product:

$$\begin{pmatrix} 4 \\ 2 \\ 6 \end{pmatrix} \odot \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix} = \begin{pmatrix} 4 \\ 0 \\ 12 \end{pmatrix}$$

forget gate controls how much of the previous memory to keep

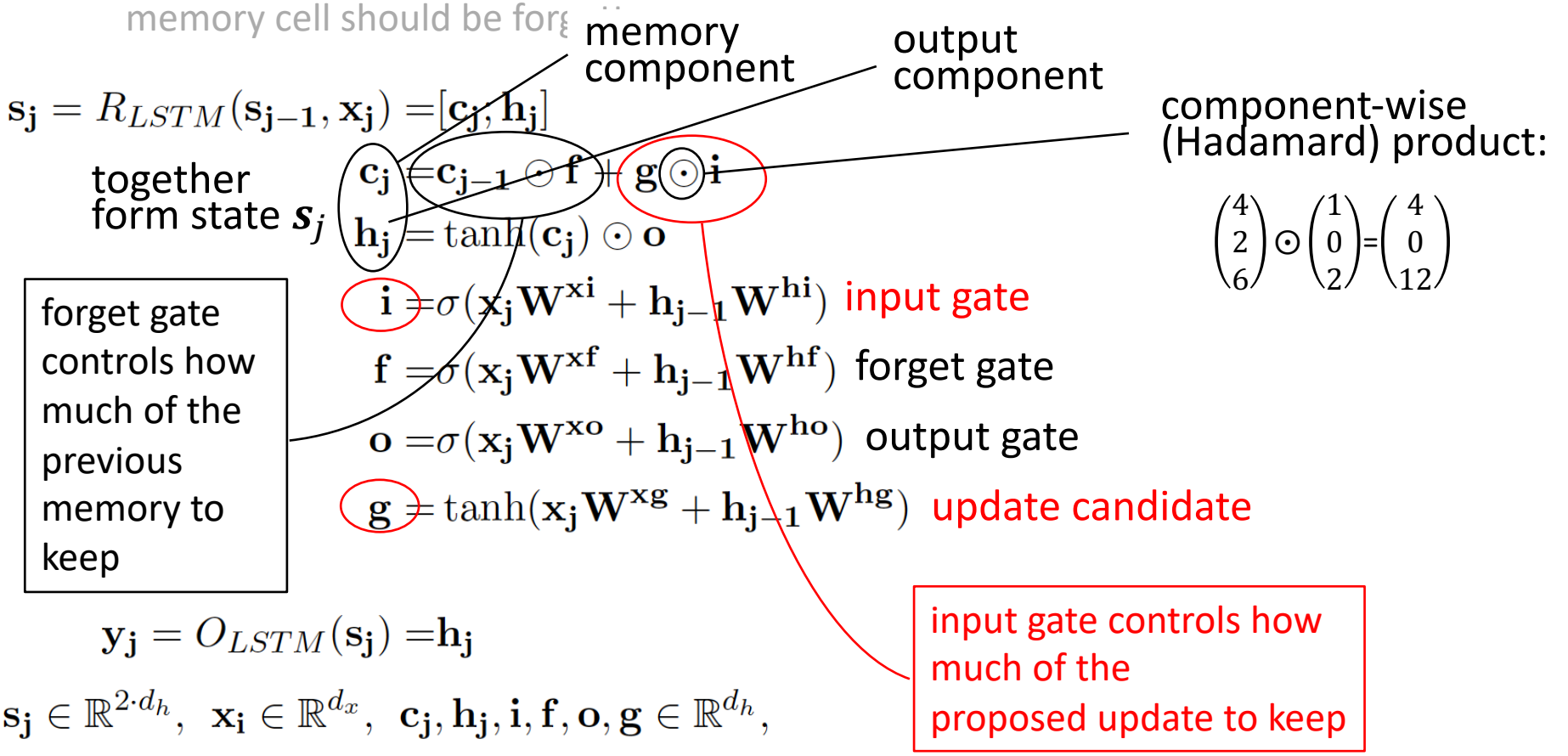
$$\mathbf{y_j} = O_{LSTM}(\mathbf{s_j}) = \mathbf{h_j}$$

$$\mathbf{s_j} \in \mathbb{R}^{2 \cdot d_h}, \quad \mathbf{x_i} \in \mathbb{R}^{d_x}, \quad \mathbf{c_j}, \mathbf{h_j}, \mathbf{i}, \mathbf{f}, \mathbf{o}, \mathbf{g} \in \mathbb{R}^{d_h},$$

$$\mathbf{W^{xo}} \in \mathbb{R}^{d_x \times d_h}, \quad \mathbf{W^{ho}} \in \mathbb{R}^{d_h \times d_h}$$

LSTMs

- straightforward RNNs: BPTT → vanishing gradients over time → hard for RNN to learn long range dependencies
- → control memory with the help of gates: decide how much of the new input should be written to the memory cell, and how much of the current content of the memory cell should be forg...



LSTMs: Practical Considerations

- recommendation: initialize bias term of forget gate close to one

- apply drop-out only on non-recurrent connections = only between layers and not between sequence positions.
  otherwise the gates cannot properly fulfil their memory control functions across time

Gated Recurrent Units (GRUs)

GRU: simplified variant of LSTM

$$\begin{aligned}\mathbf{s_j} = R_{GRU}(\mathbf{s_{j-1}}, \mathbf{x_j}) =& (\mathbf{1} - \mathbf{z}) \odot \mathbf{s_{j-1}} + \mathbf{z} \odot \mathbf{h} \\ \mathbf{z} =& \sigma(\mathbf{x_j W^{xz}} + \mathbf{h_{j-1} W^{hz}}) \\ \mathbf{r} =& \sigma(\mathbf{x_j W^{xr}} + \mathbf{h_{j-1} W^{hr}}) \\ \mathbf{h} =& \tanh(\mathbf{x_j W^{xh}} + (\mathbf{h_{j-1}} \odot \mathbf{r}) \mathbf{W^{hg}})\end{aligned}$$

$$\mathbf{y_j} = O_{LSTM}(\mathbf{s_j}) = \mathbf{s_j}$$

$$\mathbf{s_j} \in \mathbb{R}^{d_h}, \ \mathbf{x_i} \in \mathbb{R}^{d_x}, \ \mathbf{z}, \mathbf{r}, \mathbf{h} \in \mathbb{R}^{d_h}, \ \mathbf{W^{xo}} \in \mathbb{R}^{d_x \times d_h}, \ \mathbf{W^{ho}} \in \mathbb{R}^{d_h \times d_h},$$

- gate $\boldsymbol{r}$: control access to the previous state $\boldsymbol{s}_{j-1}$
- $\boldsymbol{h}$: proposed update
- updated state $\boldsymbol{s}_i$ (also serves as the output $\boldsymbol{y}_i$) determined based on an interpolation of the previous state $\boldsymbol{s}_{j-1}$ and the proposal $\boldsymbol{h}$
- gate $\boldsymbol{z}$: controls proportions of the interpolation

Other Variants

- instead of learning the amount of forgetting and remembering via controlled gates, in a simple RNN, explicitly design the $\boldsymbol{c_i}$ components of a split state $\boldsymbol{s}_i = [\boldsymbol{c}_i, \boldsymbol{h_i}]$ as slow changing context units dedicated to remembering long term dependencies via simple interpolation:

$$\mathbf{c_i} = (1-\alpha)\mathbf{x_i}\mathbf{W^{x1}} + \alpha\mathbf{c_{i-1}}$$

and a fast changing component

$$\mathbf{h_i} = \sigma(\mathbf{x_i}\mathbf{W^{x2}} + \mathbf{h_{i-1}}\mathbf{W^{h}} + \mathbf{c_i}\mathbf{W^{c}})$$

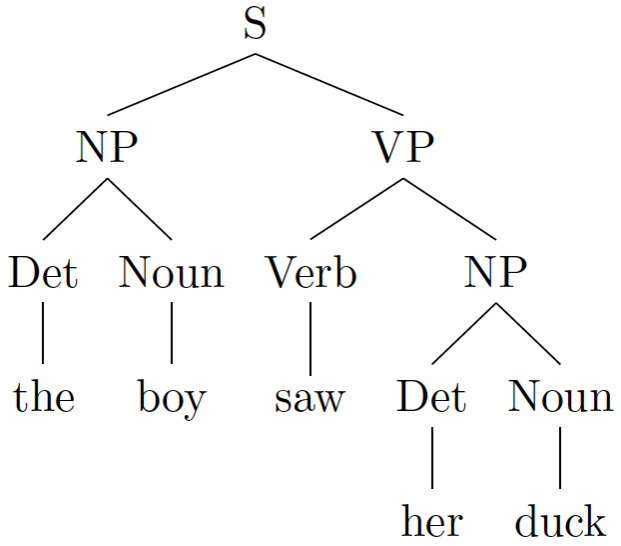
→ comparable perplexity performance in language modelling to an LSTM based approach

- additional idea: in a simple RNN, initialize the state-to-state weight matrix $\boldsymbol{W^s}$ to identity and the corresp. biases to zero → initially mostly copy the state from one time step to another → also comparable performance to LSTM-based RNN

Recursive NN (RecNN)

- generalization of RNN from sequences to trees:
  RNN encodes each sentence-prefix as a state vector
  RecNN encodes each subtree as a state vector

- binary parse tree $\mathcal{T}$ over a (word)sequence $x_{1:n}$: encode node $q_{i:j}^{A}$ representing sub-sequence $x_{i:j}$ as (A, B, C, i, k, j) or $(A \rightarrow BC, i, k, j)$ where A, B, C are the labels of subsequences $x_{i:j}, x_{i:k}, x_{k:j}$.
  (leaf nodes (←→ terminals): (A, A, A, i, i, i))



Unlabeled	Labeled	Corresponding Span
(1,1,1)	(Det, Det, Det, 1, 1, 1)	$x_{1:1}$ the
(2,2,2)	(Noun, Noun, Noun, 2, 2, 2)	$x_{2:2}$ boy
(3,3,3)	(Verb, Verb, Verb, 3, 3, 3)	saw
(4,4,4)	(Det, Det, Det, 4, 4, 4)	her
(5,5,5)	(Noun, Noun, Noun, 5, 5, 5)	duck
(4,4,5)	(NP, Det, Noun, 4, 4, 5)	her duck
(3,3,5)	(VP, Verb, NP, 3, 3, 5)	saw her duck
(1,1,2)	(NP, Det, Noun, 1, 1, 2)	the boy
(1,2,5)	(S, NP, VP, 1, 2, 5)	the boy saw her duck

Recursive NN (RecNN)

RecNN: state vectors $\boldsymbol{s}^{\mathbf{A}}_{i:j} \in \mathbb{R}^d$ represent sub-trees rooted at tree nodes $q^A_{i:j} = (A, B, C, i, k, j)$ via recurring to the state vectors $\boldsymbol{s}^{\mathbf{B}}_{i:k} \in \mathbb{R}^d$, $\boldsymbol{s}^{\mathbf{C}}_{k+1:j} \in \mathbb{R}^d$ of its children:

$$
\begin{aligned}
RecNN(x_1, \ldots, x_n, \mathcal{T}) = & \{\mathbf{s}^{\mathbf{A}}_{\mathbf{i:j}} \in \mathbb{R}^d \mid q^A_{i:j} \in \mathcal{T}\} \\
\mathbf{s}^{\mathbf{A}}_{\mathbf{i:i}} = & v(x_i) \\
\mathbf{s}^{\mathbf{A}}_{\mathbf{i:j}} = & R(A, B, C, \mathbf{s}^{\mathbf{B}}_{\mathbf{i:k}}, \mathbf{s}^{\mathbf{C}}_{\mathbf{k+1:j}}) \qquad q^B_{i:k} \in \mathcal{T}, \;\; q^C_{k+1:j} \in \mathcal{T}
\end{aligned}
$$

with R: NN style linear transformation followed by activation (ignoring the explicit labels A,B,C):

$$R(A, B, C, \mathbf{s}^{\mathbf{B}}_{\mathbf{i:k}}, \mathbf{s}^{\mathbf{C}}_{\mathbf{k+1:j}}) = g([\mathbf{s}^{\mathbf{B}}_{\mathbf{i:k}}; \mathbf{s}^{\mathbf{C}}_{\mathbf{k+1:j}}]\mathbf{W})$$

or incorporating the labels (in the form of d_{nt}-dim. label embeddings):

$$R(A, B, C, \mathbf{s}^{\mathbf{B}}_{\mathbf{i:k}}, \mathbf{s}^{\mathbf{C}}_{\mathbf{k+1:j}}) = g([\mathbf{s}^{\mathbf{B}}_{\mathbf{i:k}}; \mathbf{s}^{\mathbf{C}}_{\mathbf{k+1:j}}; v(A); v(B); v(C)]\mathbf{W})$$

$\mathbf{W} \in \mathbb{R}^{2d+2d_{nt} \times d}$

Recursive NN (RecNN)

- other variant (especially for small (e.g. phrase structure) grammars): do not share weights but make them specific to productions:

$$R(A, B, C, \mathbf{s^B_{i:k}}, \mathbf{s^C_{k+1:j}}) = g([\mathbf{s^B_{i:k}}; \mathbf{s^C_{k+1:j}}]\mathbf{W^{BC}})$$

- Training RecNNs: as usual ☺
  Loss: either at root node only or at each node (combine losses via summation as usual)

- possible: use states (representation of sub-trees) as "embeddings" of that sub-tree and pass them as input to other NNs / other layers (for classification etc.)

Bibliography

(1) Yoav Goldberg: “A Primer on Neural Network Models for Natural Language Processing” http://www.cs.biu.ac.il/~yogo/nnlp.pdf (URL, May 2018) and at https://arxiv.org/abs/1510.00726v1 (URL, May 2018), 2015

(2) Richard Socher et al: “CS224n: Natural Language Processing with Deep Learning”, Lecture Materials Winter 2018 (slides and links to background reading) https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1184 (URL, Oct 2019), 2018

(3) Pennington, J., Socher, R. and Manning, C., 2014. Glove: Global vectors for word representation. In Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP) (pp. 1532-1543).

(4) Goodfellow, Ian, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. Deep learning. Vol. 1. Textbook, Cambridge: MIT press, 2016.

(5) Günnemann, S., Shchur, A., Bojchevski, A., Groh, G.: IN2064 Machine Learning I: Lecture, Exercises, Practical Sessions; TUM Winter Term 2017/18

(6) Yoshua Bengio: Understanding and Improving Deep Learning Algorithms, MLGoogle Distinguished Lecture, 2010 (in [5])

Bibliography

(7) Levy, Omer, and Yoav Goldberg. "Dependency-based word embeddings." *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Vol. 2. 2014.

(8) Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747(2016).

(9) Goldberg, Yoav, and Omer Levy. "word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method." arXiv preprint arXiv:1402.3722 (2014).

(10) Christopher Manning et al: “CS224n: Natural Language Processing with Deep Learning” Winter 2019, Lecture Materials (slides and links to background reading) http://web.stanford.edu/class/cs224n/ (URL, Oct 2019), 2019

Recommendations for Studying

- minimal approach:

  work with the slides and understand their contents! Think beyond instead of merely memorizing the contents

- standard approach:

  minimal approach + read [1]. Study the corresponding lecture slides from [2] for additional details omitted in our slides

- interested student's approach:

  standard approach + read [3]

- deeply interested student's approach:

  standard approach + read all of the recommended background reading of [2] from lecture 1 up to lecture 9