TTIC 31210:
Advanced Natural Language Processing

Kevin Gimpel

Spring 2019

Lecture 4:
Subword Modeling and Contextualized Word Embeddings

Roadmap

- intro (1 lecture)
- **deep learning for NLP (5 lectures)**
- structured prediction: sequence labeling, syntactic and semantic parsing, dynamic programming (4 lectures)
- generative models, latent variables, unsupervised learning, variational autoencoders (2 lectures)
- Bayesian methods in NLP (2 lectures)
- Bayesian nonparametrics in NLP (2 lectures)
- review & other topics (1 lecture)

Today

- modeling subword structure in words
- contextualized word embeddings

Recap

- on Monday we briefly reviewed some models and loss functions for word embeddings

Other Work on Word Embeddings

- using subword information (e.g., characters) in word embeddings

- multiple embeddings for a single word type corresponding to different word senses

- tailoring embeddings using particular resources or for particular NLP tasks

Other Work on Word Embeddings

- using subword information (e.g., characters) in word embeddings

- multiple embeddings for a single word type corresponding to different word senses

- tailoring embeddings using particular resources or for particular NLP tasks

Subword Modeling for Word Embeddings

- Using word embeddings has limitations:
  - closed vocabulary (100k-300k words is typical)
  - large number of parameters! (100k * 300)
  - for morphologically-rich languages, using a separate vector for each word type is “obviously” wrong

- Solution: **character**-level modeling
  - open vocabulary, fewer parameters, often similar or better performance

Early Neural Methods

morphological analyzer + recursive neural network:

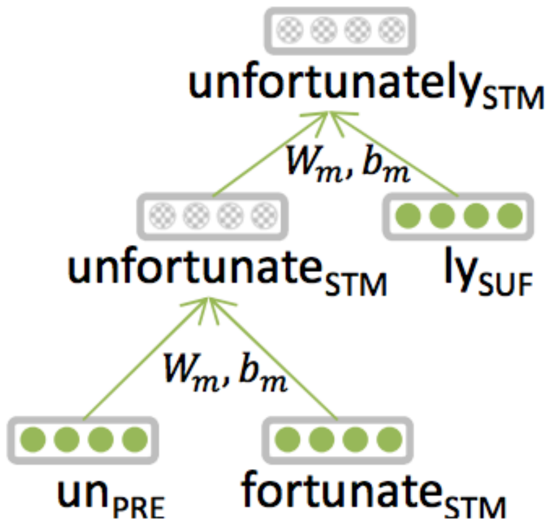


Figure 1: **Morphological Recursive Neural Network**. A vector representation for the word "unfortunately" is constructed from morphemic vectors: $un_{\texttt{pre}}$, $fortunate_{\texttt{stm}}$, $ly_{\texttt{suf}}$. Dotted nodes are computed on-the-fly and not in the lexicon.

Luong et al. (2013): *Better Word Representations with Recursive Neural Networks for Morphology*

unsupervised morphological analysis & vector addition:

$$\overrightarrow{\text{imperfection}} = \overrightarrow{im} + \overrightarrow{perfect} + \overrightarrow{ion}$$
$$\overrightarrow{\text{perfectly}} = \overrightarrow{perfect} + \overrightarrow{ly}.$$

We include the surface form of a word as a factor itself. This accounts for noncompositional constructions ($\overrightarrow{\text{greenhouse}} = \overrightarrow{greenhouse} + \overrightarrow{green} + \overrightarrow{house}$), and makes the approach more robust to noisy morphological segmentation. This strategy also overcomes the order-invariance of additive composition ($\overrightarrow{\text{hangover}} \neq \overrightarrow{\text{overhang}}$).

Botha & Blunsom (2014): *Compositional Morphology for Word Representations and Language Modelling*

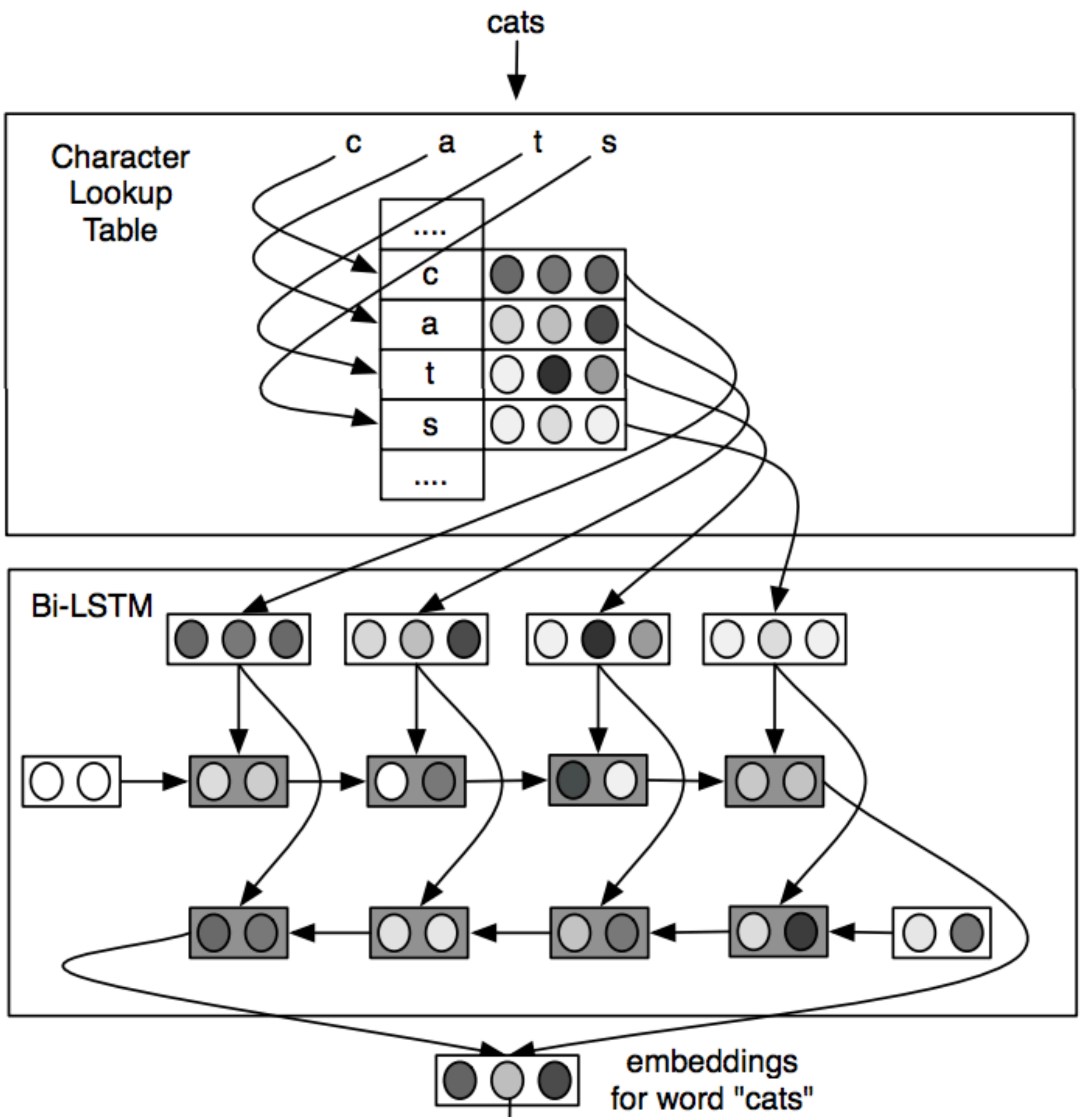
visualization of learned morpheme vectors:

Figure 6. **English morpheme vectors learnt by CLBL++.**

Botha & Blunsom (2014): *Compositional Morphology for Word Representations and Language Modelling*

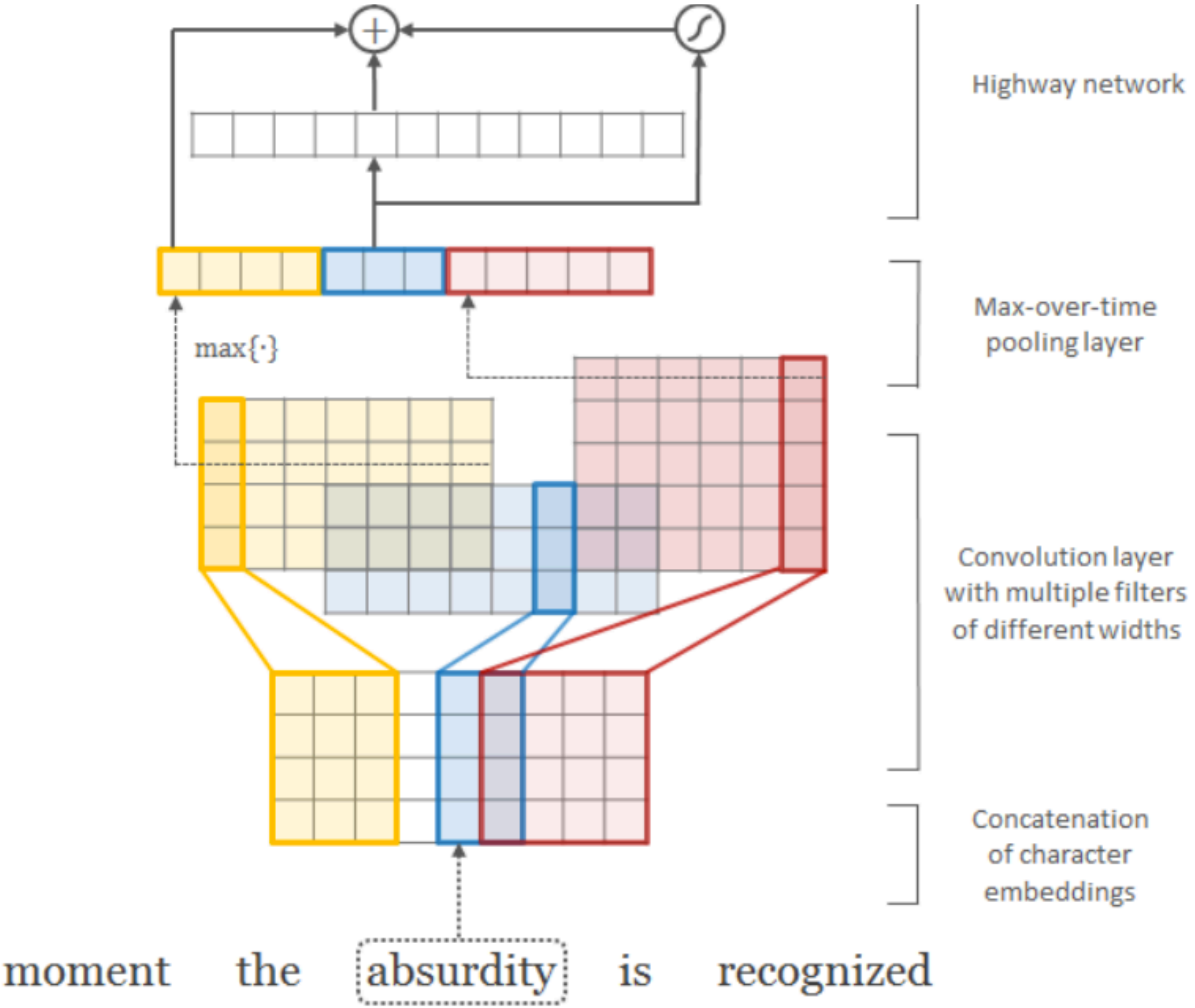
- 2013-2014: morphological analyzers + define composition function on morphemes + learn embeddings for morphemes
- today, researchers use one of the following:
  - RNNs on character sequences (Ling et al., 2015; Ballesteros et al., 2015)
  - CNNs on character sequences (dos Santos and Zadrozny, 2014; Zhang et al., 2015; Kim et al., 2016)
  - represent words as bags of character *n*-grams, learn embeddings for character *n*-grams

Bidirectional LSTM over Characters



Ling et al. (2015): *Finding Function in Form: Compositional Character Models for Open Vocabulary Word Representation*

Convolutional Neural Network over Character Sequence



Kim et al. (2016): *Character-Aware Neural Language Models*

Convolutional Neural Networks

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

Filters

- for now, think of a filter as a vector in the word embedding space

- the filter matches a particular region of the space

- “match” = “has high dot product with”

Convolution

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

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

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

Convolution

compute dot product of filter and each word vector:

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

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

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

Convolution

compute dot product of filter and each word vector:

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

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

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

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

Convolution

compute dot product of filter and each word vector:

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

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

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

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

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

Convolution

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

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

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

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

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

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

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

Convolution

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

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

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

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

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

$\mathbf{c}$ = "feature map" for this filter,

has an entry for each position in input (in this case, 3 entries)

Pooling

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

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

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

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

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

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

Pooling

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

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

use **pooling**:

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

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

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

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

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

$\mathbf{c}$ es)

Convolutional Neural Networks

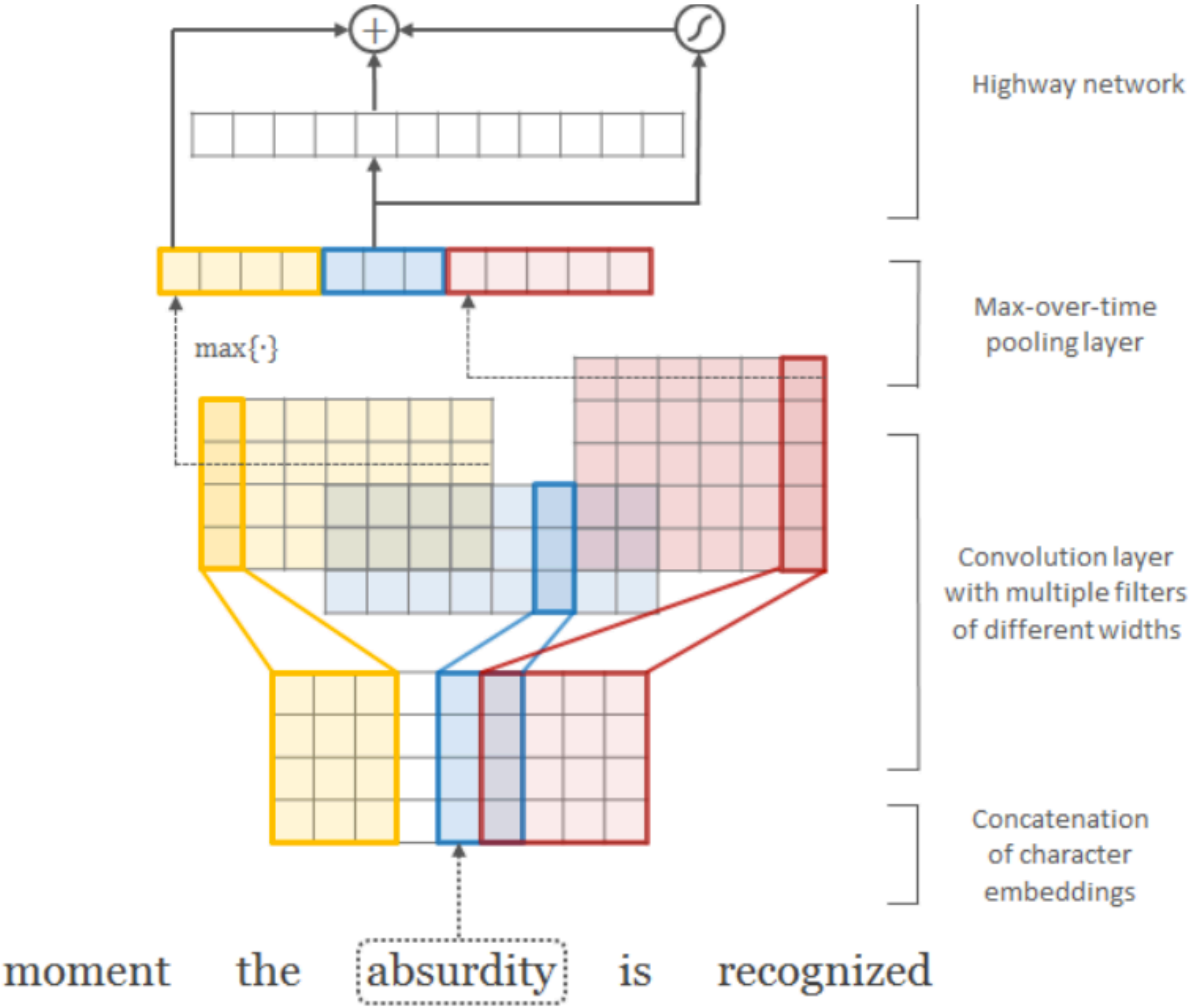
- "convolutional layer" = set of filters that are convolved with the input vector (whether ***x*** or hidden vector)

- could be followed by more convolutional layers, or by a type of pooling

- filters of varying n-gram lengths commonly used (1- to 5-grams)

- CNNs commonly used for character-level processing; filters look at character n-grams

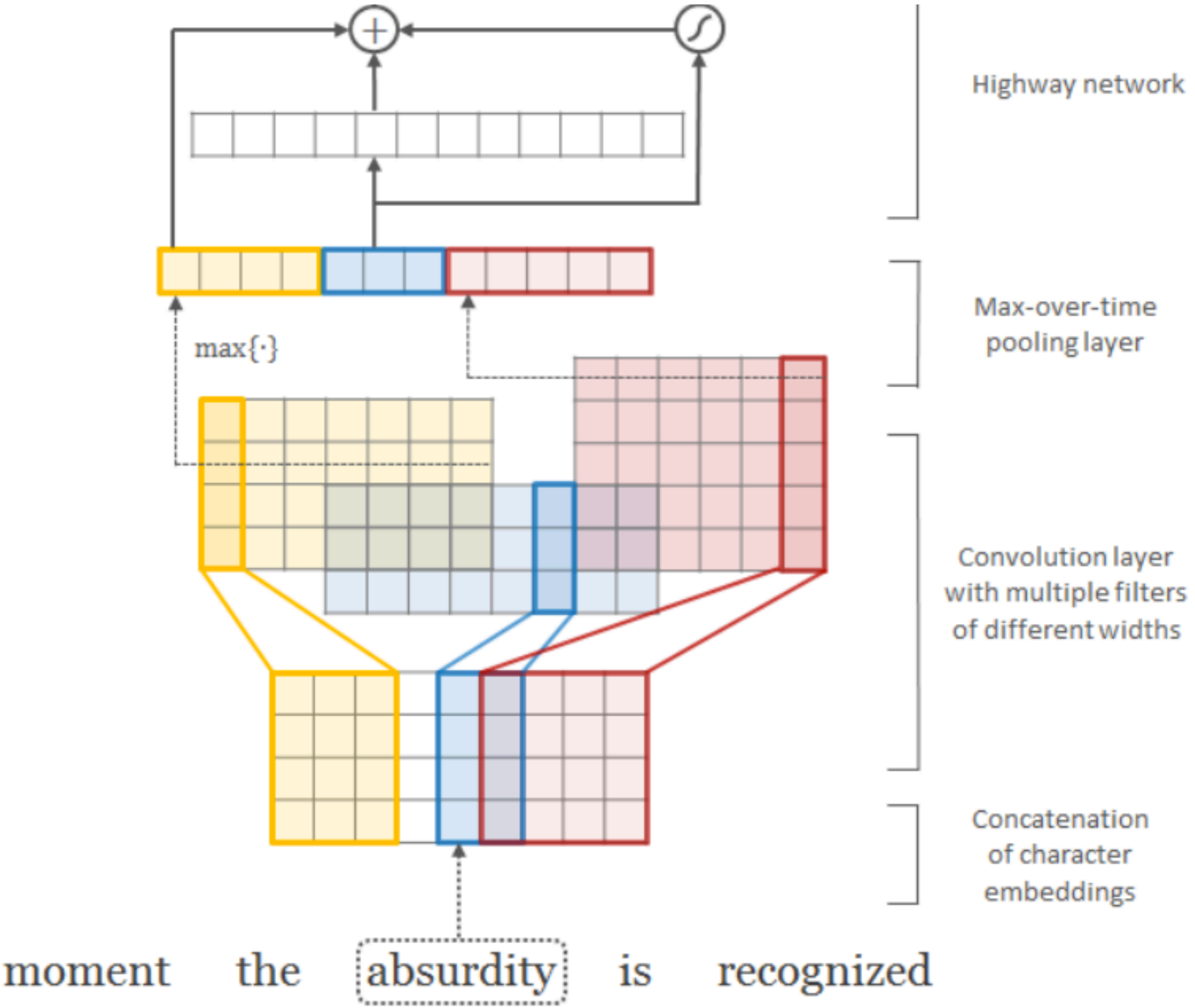
Convolutional Neural Network over Characters



Kim et al. (2016): *Character-Aware Neural Language Models*

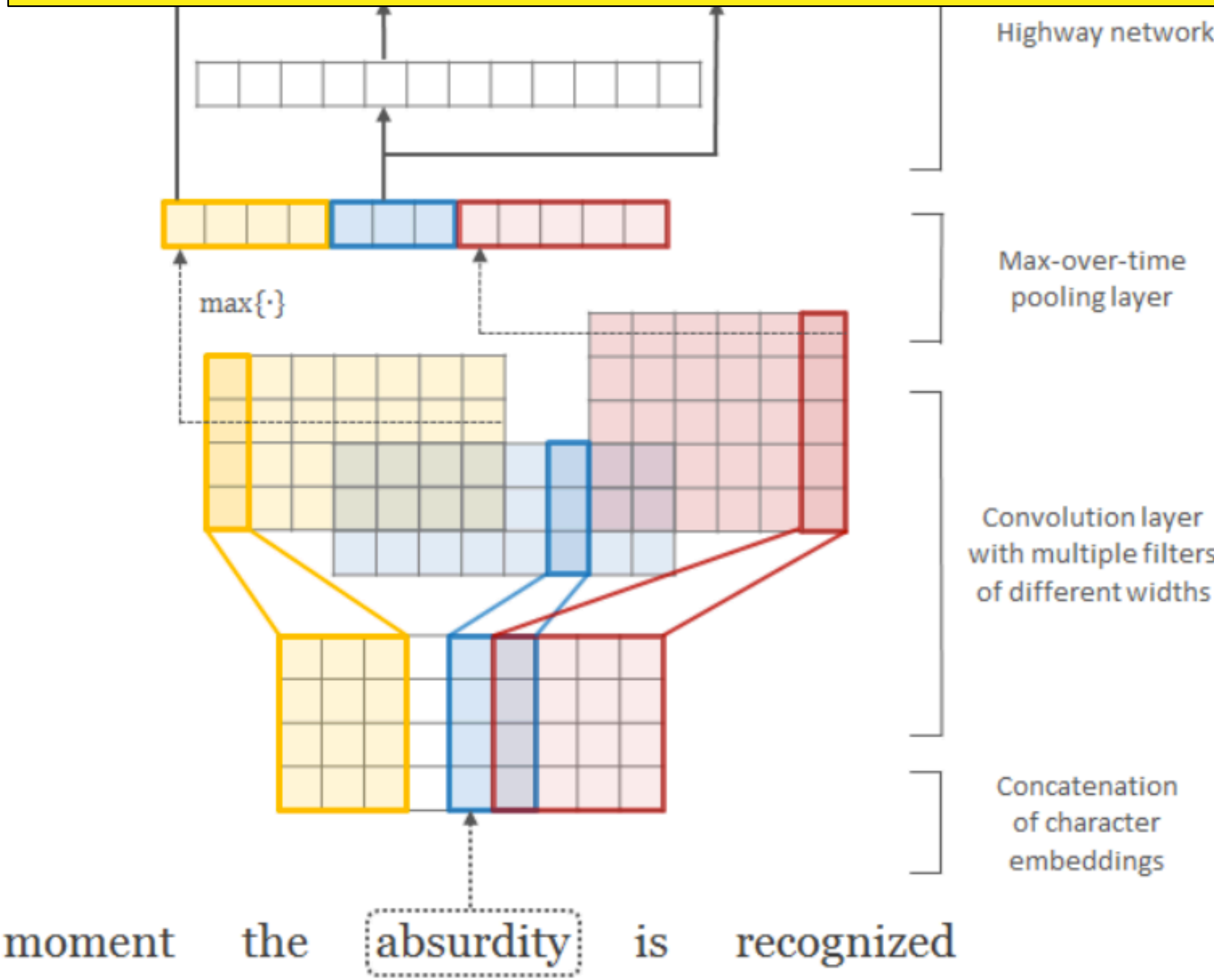
1. What dimension are the character embeddings?

ters



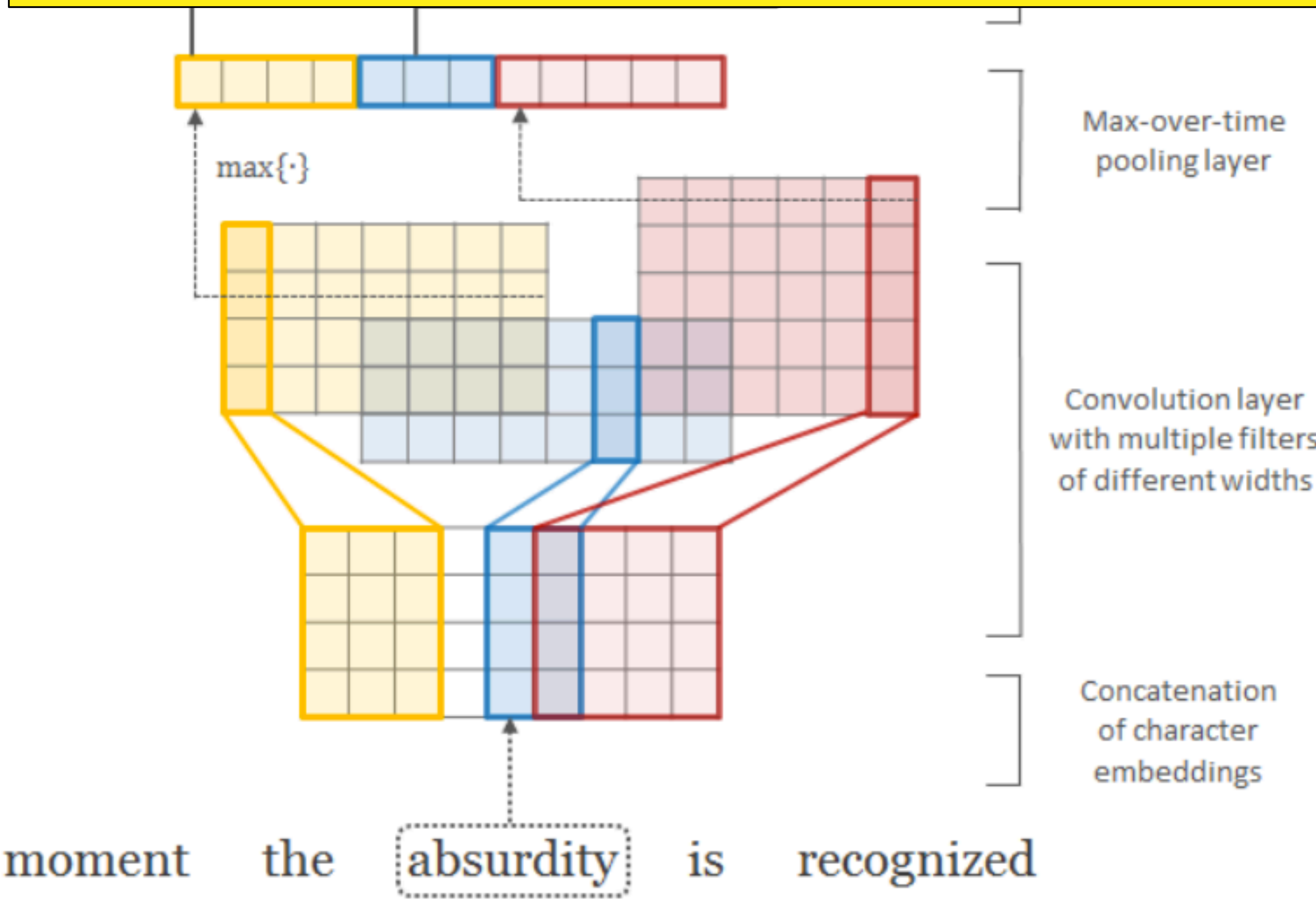
Kim et al. (2016): *Character-Aware Neural Language Models*

1. What dimension are the character embeddings? **4**
2. How many character 4-gram filters are there?



Kim et al. (2016): *Character-Aware Neural Language Models*

1. What dimension are the character embeddings? **4**
2. How many character 4-gram filters are there? **5**
3. Why do different filter lengths lead to different lengths of feature maps?



Kim et al. (2016): *Character-Aware Neural Language Models*

- what about simpler methods?
- add or average vectors for character *n*-grams in the word:
  - word space (Schutze, 1993)
  - deep structured semantic models (Huang et al., 2013)
  - charagram (Wieting et al., 2016)
  - fastText (Bojanowski et al., 2017)

DSSM (Microsoft Research, 2013-2016)

Tri-letter: a scale-able word representation

- Tri-letter based Word Hashing of "cat"
  - -> #cat#
  - Tri-letters: #-c-a, c-a-t, a-t-#.

- Compact representation
  - |Voc| (500K) → |TriLetter| (30K)
- Generalize to unseen words
- Robust to misspelling, inflection, etc.

$$x\,(cat) = \begin{bmatrix}0\\ \vdots \\ 1 \\ \vdots \\ 0\end{bmatrix}$$

The index of word *cat* in the vocabulary

$$f\,(cat) = \begin{bmatrix}0\\ \vdots \\ 1 \\ \vdots \\ 1 \\ \vdots \\ 1 \\ \vdots \\ 0\end{bmatrix}$$

Indices of *#-c-a*, *c-a-t*, *a-t-#* in the letter-tri-gram list, respectively.

Huang et al. (2013): *Learning Deep Structured Semantic Models for Web Search using Clickthrough Data*

DSSM (Microsoft Research, 2013-2016)

Word hashing by n-gram of letters

- Collision:
  - What if different words have the same word hashing vector?
  - Statistics
    - 22 out of 500K words collide
    - Collision Example: #bananna# <- > #bannana#

Vocabulary size	Unique tri-letter observed in voc	Number of Collisions
40K	10306	2
500K	30621	22

Huang et al. (2013): *Learning Deep Structured Semantic Models for Web Search using Clickthrough Data*

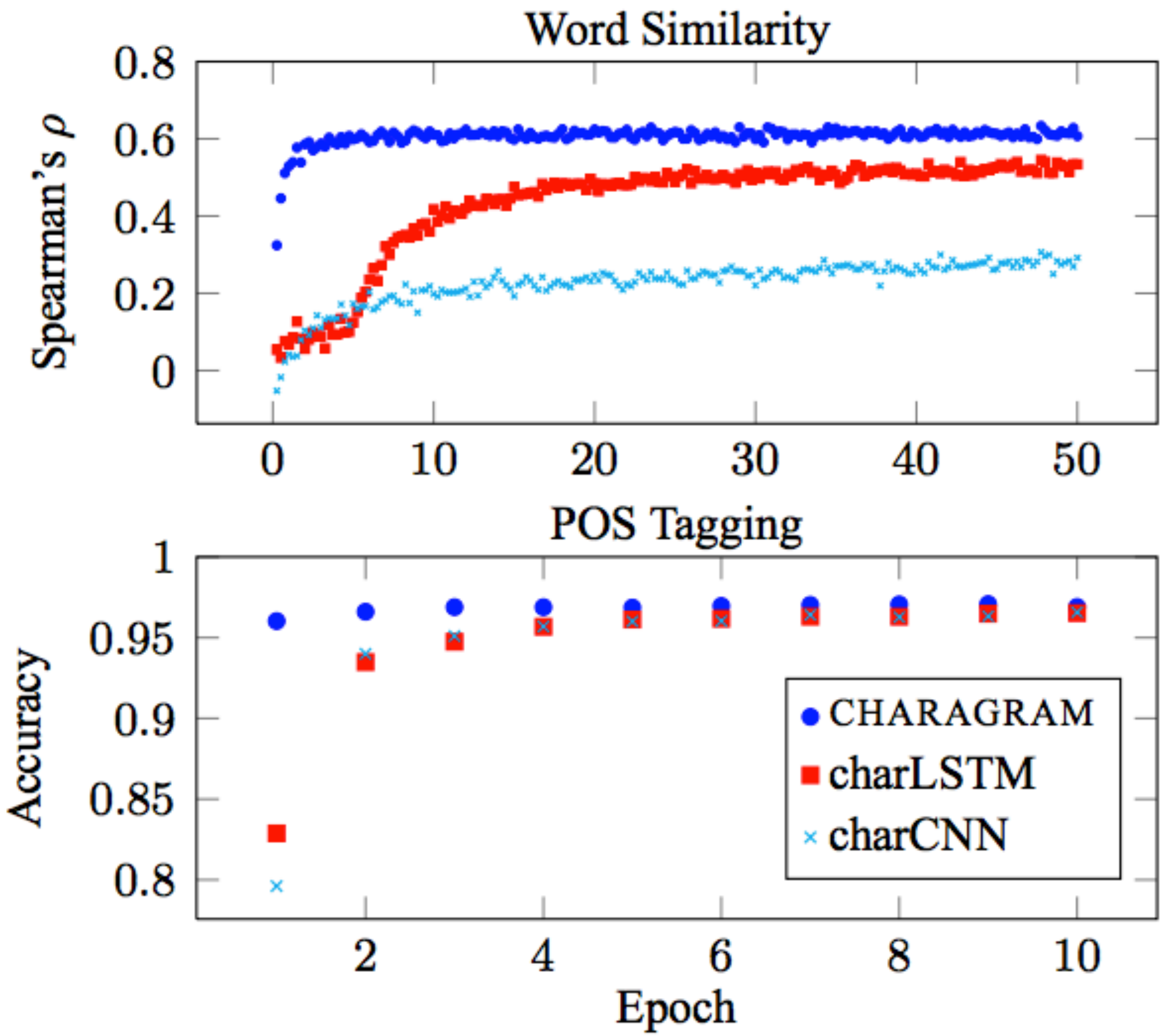
“Charagram” Embeddings

- to embed a character sequence (word or sentence), sum embeddings for character *n*-grams

- only parameters to learn are embeddings for character *n*-grams

Wieting et al. (2016): *Charagram: Embedding Words and Sentences via Character n-grams*

Charagram Embeddings



faster convergence to strong performance than character LSTM or CNN

Wieting et al. (2016): *Charagram: Embedding Words and Sentences via Character n-grams*

Charagram Word Embeddings

- we used all 122,610 character n-grams observed in training set ($2 \leq n \leq 4$), including spaces

- we trained on paraphrase pairs from the Paraphrase Database

Wieting et al. (2016): *Charagram: Embedding Words and Sentences via Character n-grams*

For words in training set:

word	nearest neighbors
refunding	refunds, refunded, refund, repayment, reimbursement, rebate, repay reimbursements, reimburse, repaying, repayments, rebates, rebating
professors	professor, professorships, professorship, teachers, professorial, teacher prof., teaches, lecturers, teachings, instructors, headteachers
huge	enormous, tremendous, large, big, vast, overwhelming, immense, giant formidable, considerable, massive, huger, large-scale, great, daunting

Wieting et al. (2016): *Charagram: Embedding Words and Sentences via Character n-grams*

For words in training set:

word	nearest neighbors
refunding	refunds, refunded, refund, repayment, reimbursement, rebate, repay reimbursements, reimburse, repaying, repayments, rebates, rebating
professors	professor, professorships, professorship, teachers, professorial, teacher prof., teaches, lecturers, teachings, instructors, headteachers
huge	enormous, tremendous, large, big, vast, overwhelming, immense, giant formidable, considerable, massive, huger, large-scale, great, daunting

For words not in training set:

word	nearest neighbors
vehicals	vehical, vehicles, vehicels, vehicular, cars, vehicle, automobiles, car
journeying	journey, journeys, voyage, trip, roadtrip, travel, tourney, voyages, road-trip
babyyyyyy	babyyyyyyy, baby, babys, babe, baby.i, babydoll, babycake, darling

Wieting et al. (2016): *Charagram: Embedding Words and Sentences via Character n-grams*

fastText

- like word2vec, but represents a word as the sum of its character *n*-gram embeddings and an embedding for the word itself

We also include the word w itself in the set of its n-grams, to learn a representation for each word (in addition to character n-grams). Taking the word *where* and $n = 3$ as an example, it will be represented by the character n-grams:

<wh, whe, her, ere, re>

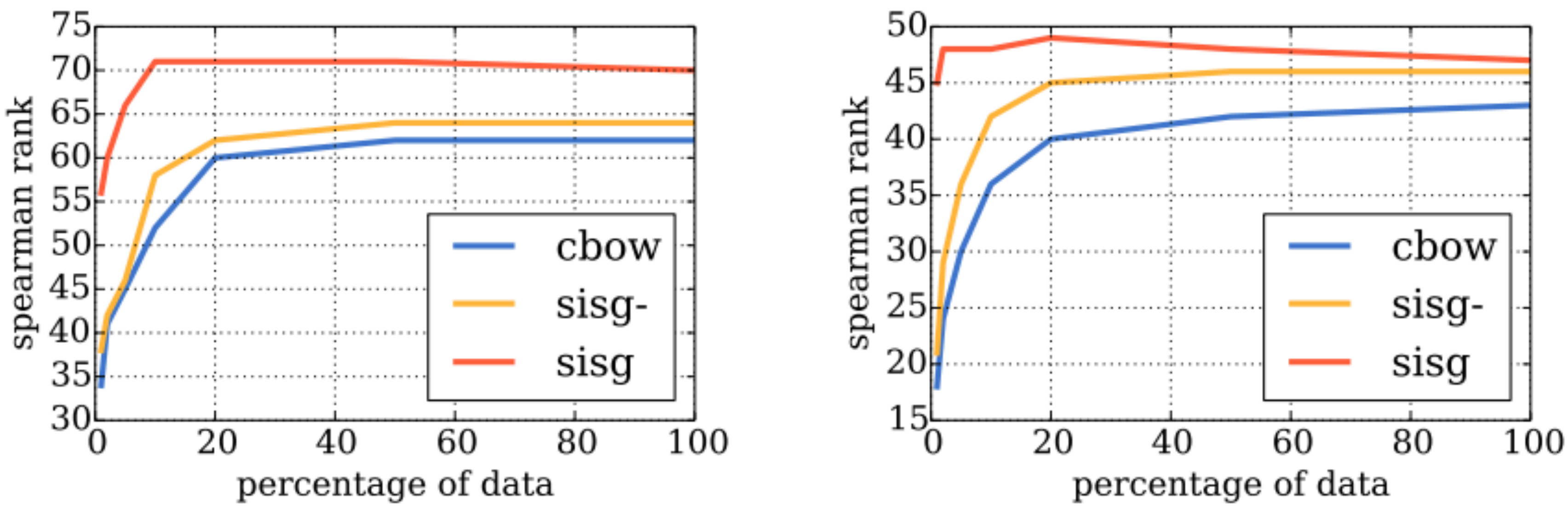
and the special sequence

<where>.

Bojanowski et al. (2017): *Enriching Word Vectors with Subword Information*

fastText

- better data efficiency than word2vec:



(a) DE-GUR350 (b) EN-RW

Figure 1: Influence of size of the training data on performance. We compute word vectors following the proposed model using datasets of increasing size. In this experiment, we train models on a fraction of the full Wikipedia dump.

Bojanowski et al. (2017): *Enriching Word Vectors with Subword Information*

- if you're just encoding text (rather than generating), you can use neural architectures like these to capture subword information

- for generation, it's trickier:
  - character RNNs are fine for generating words, but not sentences (very long sequences and long-distance dependencies)

- simple, data-driven segmentation methods have emerged as the standard way to handle this

Data-Driven Segmentation

- Most popular methods:
  - Byte pair encoding (BPE)
  - SentencePiece's unigram LM

SentencePiece

build passing | build passing | coverage 98% | issues 10 open | code quality A | pypi package 0.1.81 | contributions welcome | License Apache 2.0

SentencePiece is an unsupervised text tokenizer and detokenizer mainly for Neural Network-based text generation systems where the vocabulary size is predetermined prior to the neural model training. SentencePiece implements **subword units** (e.g., **byte-pair-encoding (BPE)** [Sennrich et al.]) and **unigram language model** [Kudo.]) with the extension of direct training from raw sentences. SentencePiece allows us to make a purely end-to-end system that does not depend on language-specific pre/postprocessing.

This is not an official Google product.

Technical highlights

Data-Driven Segmentation

- Most popular methods:
  - Byte pair encoding (BPE)
  - SentencePiece's unigram LM

- these are easy and fast to use & work well

- they permit unbounded vocabularies with a relatively small number of parameters

Byte Pair Encoding (BPE)
(Gage, 1994)

- simple data compression technique
- iteratively replaces most frequent pair of bytes in a sequence with a single, unused byte
- Sennrich et al. (2016) adapted BPE for characters and character sequences

Byte Pair Encoding (BPE)

- **merge**: a rule that combines two consecutive units into a single unit
- initially, units are characters
- after merges, units become character sequences
- greedy algorithm:
  - merge two units with the largest unit bigram count, produce merged unit
  - replace all instances of that 2-unit sequence with the merged unit, recompute counts

Byte Pair Encoding (BPE)

- Sennrich et al. use BPE based on word counts from a corpus
  - sentences are not used; all that's needed are word types and their counts
  - special treatment for end-of-word symbol </w> (an unseen initial step merges final character in each word with </w>)
  - when segmenting new data, segments words individually (does not use context)

Corpus:

```
low
low
lower
lowest
high
high
higher
```

Merges:

Sennrich et al. (2016): *Neural Machine Translation of Rare Words with Subword Units*

Corpus:

low</w>

low</w>

lower</w>

lowest</w>

high</w>

high</w>

higher</w>

Merges:

actually the corpus looks like this

Sennrich et al. (2016): *Neural Machine Translation of Rare Words with Subword Units*

Corpus:

low</w>

low</w>

lower</w>

lowest</w>

high</w>

high</w>

higher</w>

Merges:

w </w> (2)

r </w> (2)

h </w> (2)

t </w> (1)

the first thing we do is merge word-ending characters with </w>

Sennrich et al. (2016): *Neural Machine Translation of Rare Words with Subword Units*

Corpus: Segmentation: Merges:

```
low</w>
low</w>
lower</w>
lowest</w>
high</w>
high</w>
higher</w>
```

```
w </w> (2)
r </w> (2)
h </w> (2)
t </w> (1)
```

given this set of merges, let's segment the corpus!

Sennrich et al. (2016): *Neural Machine Translation of Rare Words with Subword Units*

Corpus:	Segmentation:
low</w>	l o w</w>
low</w>	l o w</w>
lower</w>	l o w e r</w>
lowest</w>	l o w e s t</w>
high</w>	h i g h</w>
high</w>	h i g h</w>
higher</w>	h i g h e r</w>

Merges:

w </w> (2)
r </w> (2)
h </w> (2)
t </w> (1)

Sennrich et al. (2016): *Neural Machine Translation of Rare Words with Subword Units*

Corpus:	Segmentation:	Merges:
`low</w>`	`l o w</w>`	`w </w> (2)`
`low</w>`	`l o w</w>`	`r </w> (2)`
`lower</w>`	`l o w e r</w>`	`h </w> (2)`
`lowest</w>`	`l o w e s t</w>`	`t </w> (1)`
`high</w>`	`h i g h</w>`	
`high</w>`	`h i g h</w>`	
`higher</w>`	`h i g h e r</w>`	

re

looking at the segmented corpus shows us what merge will occur next

Corpus:	Segmentation:	Merges:
low</w>	l o w</w>	w </w> (2)
low</w>	l o w</w>	r </w> (2)
lower</w>	l o w e r</w>	h </w> (2)
lowest</w>	l o w e s t</w>	t </w> (1)
high</w>	h i g h</w>	
high</w>	h i g h</w>	
higher</w>	h i g h e r</w>	

what is the next merge?
(what unit bigram appears most often?)

Sennric … re
Words with Subword Units

Corpus:	Segmentation:	Merges:
low</w>	l o w</w>	w </w> (2)
low</w>	l o w</w>	r </w> (2)
lower</w>	l o w e r</w>	h </w> (2)
lowest</w>	l o w e s t</w>	t </w> (1)
high</w>	h i g h</w>	l o (4)
high</w>	h i g h</w>	
higher</w>	h i g h e r</w>	

Sennrich et al. (20
Words with Subw

given this new set of merges, let’s re-segment the corpus!

Corpus:	Segmentation:	Merges:
low</w>	lo w</w>	w </w> (2)
low</w>	lo w</w>	r </w> (2)
lower</w>	lo w e r</w>	h </w> (2)
lowest</w>	lo w e s t</w>	t </w> (1)
high</w>	h i g h</w>	l o (4)
high</w>	h i g h</w>	
higher</w>	h i g h e r</w>	

Sennrich et al. (2016): *Neural M*
Words with Subword Units

note: we will always "back off" to the complete segmentation

Corpus:	Segmentation:	Merges:
`low</w>`	`lo w</w>`	`w </w> (2)`
`low</w>`	`lo w</w>`	`r </w> (2)`
`lower</w>`	`lo w e r</w>`	`h </w> (2)`
`lowest</w>`	`lo w e s t</w>`	`t </w> (1)`
`high</w>`	`h i g h</w>`	`l o (4)`
`high</w>`	`h i g h</w>`	
`higher</w>`	`h i g h e r</w>`	

what is the next merge?
(what unit bigram appears most often?)

Sennric ... *re*
Words

Corpus:	Segmentation:	Merges:
low</w>	lo w</w>	w </w> (2)
low</w>	lo w</w>	r </w> (2)
lower</w>	lo w e r</w>	h </w> (2)
lowest</w>	lo w e s t</w>	t </w> (1)
high</w>	h i g h</w>	l o (4)
high</w>	h i g h</w>	i g (3)
higher</w>	h i g h e r</w>	

Corpus:	Segmentation:	Merges:
low</w>	lo w</w>	w </w> (2)
low</w>	lo w</w>	r </w> (2)
lower</w>	lo w e r</w>	h </w> (2)
lowest</w>	lo w e s t</w>	t </w> (1)
high</w>	h ig h</w>	l o (4)
high</w>	h ig h</w>	i g (3)
higher</w>	h ig h e r</w>	

Corpus:	Segmentation:	Merges:
low</w>	lo w</w>	w </w> (2)
low</w>	lo w</w>	r </w> (2)
lower</w>	lo w e r</w>	h </w> (2)
lowest</w>	lo w e s t</w>	t </w> (1)
high</w>	h ig h</w>	l o (4)
high</w>	h ig h</w>	i g (3)
higher</w>	h ig h e r</w>	h ig (3)

Corpus:	Segmentation:	Merges:
low</w>	lo w</w>	w </w> (2)
low</w>	lo w</w>	r </w> (2)
lower</w>	lo w e r</w>	h </w> (2)
lowest</w>	lo w e s t</w>	t </w> (1)
high</w>	hig h</w>	l o (4)
high</w>	hig h</w>	i g (3)
higher</w>	hig h e r</w>	h ig (3)

Corpus:

low</w>
low</w>
lower</w>
lowest</w>
high</w>
high</w>
higher</w>

Segmentation:

low</w>
low</w>
lowe r</w>
lowe s t</w>
high</w>
high</w>
hig h e r</w>

Merges:

w </w> (2)
r </w> (2)
h </w> (2)
t </w> (1)
l o (4)
i g (3)
h ig (3)
w e (2)
lo we (2)
lo w</w>(2)
hig h</w>(2)

we'd like to merge e and r</w>, but we merged w and e already, so that messed us up

New Corpus:	Merges for New Corpus:	
low (2x)	l o	s t</w>
lower	s m	lo w</w>
lowest	sm a	lo w
high (2x)	sma l	hig h</w>
higher	i g	e st</w>
small	h ig	
smaller	e r</w>	
smallest	smal l	

Merges:	Application:
`l o`	`low → low`
`s m`	`lower → low er`
`sm a`	`lowest → low est`
`sma l`	`high → high`
`i g`	`higher → hig h er`
`h ig`	`highest → hig h est`
`e r</w>`	`small → smal l`
`smal l`	`smaller → small er`
`s t</w>`	`smallest → small est`
`lo w</w>`	
`lo w`	
`hig h</w>`	
`e st</w>`	

- we can limit the vocabulary size of the segmented data by limiting the number of merges

- this can be very helpful for handling an open vocabulary of words while reducing computation (e.g., when using a softmax over the vocabulary)

Stanford Sentiment Treebank

BPE merging on train+dev sets

up to 20k merges (max number found: 15,417)

Example from test set:

`writer/director/producer` →

`writ@@ er@@ /@@ direct@@ or@@ /@@ producer`

(To recover original text, remove "`@@ `")

```
writ@@ er@@ /@@ direct@@ or@@ /@@ producer
```

```
Here 's a British flick gle@@ efully un@@
concerned with plau@@ sibility , yet just
as determined to entertain you .
```

probably good: “unconcerned” becomes “un concerned”
maybe bad: “gle efully”

```
It would n't be my prefer@@ red way of
sp@@ ending 100 minutes or $ 7@@ .@@ 00 .
```

probably good: “prefer” is related to “preferred”
maybe bad: “spending” is not related to “ending”

- BPE is a useful hack, doesn't correspond to optimizing any probabilistic objective function

- other related methods have interpretations as probabilistic models

- we will see methods later in the course for unsupervised segmentation using probabilistic modeling and priors related to "minimum description length"

Other Work on Word Embeddings

- using subword information (e.g., characters) in word embeddings

- multiple embeddings for a single word type corresponding to different word senses

- tailoring embeddings using particular resources or for particular NLP tasks

Other Work on Word Embeddings

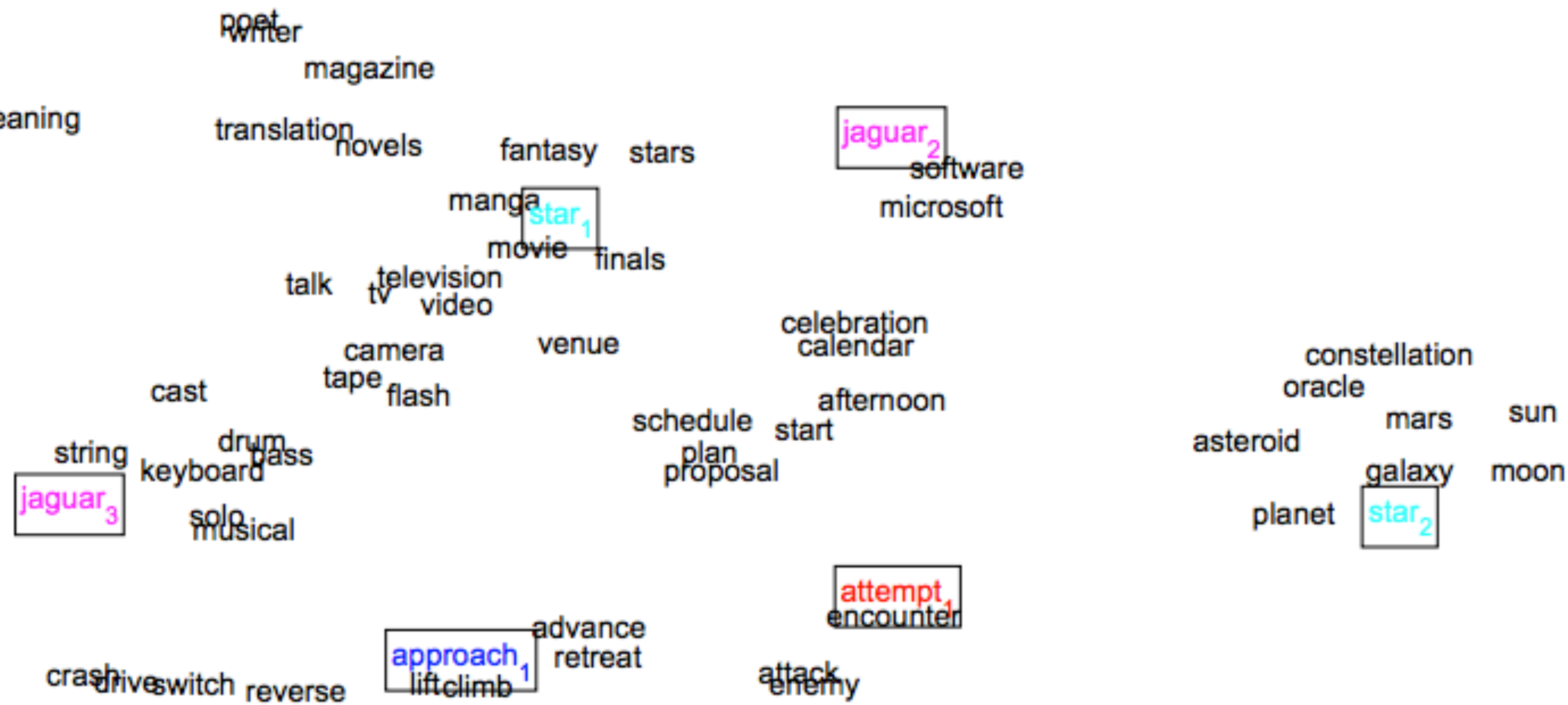
- using subword information (e.g., characters) in word embeddings
- multiple embeddings for a single word type corresponding to different word senses
- tailoring embeddings using particular resources or for particular NLP tasks

Multisense Word Embeddings

- one embedding for a word type is insufficient
  - due to different senses of a word, different meanings (polysemy, homonymy)

- there has been a lot of work in learning sense-specific word embeddings:
  - use a word sense labeler or cluster word tokens into clusters that capture word sense
  - learn embeddings for each sense/cluster

Multisense Word Embeddings



Huang et al. (2012): *Improving Word Representations Via Global Context And Multiple Word Prototypes*

- nearest neighbors given context:

Context	Nearest Neighbors
Apple is a kind of fruit.	pear, cherry, mango, juice, peach, plum, fruit, cider, apples, tomato, orange, bean, pie
Apple releases its new ipads.	microsoft, intel, dell, ipad, macintosh, ipod, iphone, google, computer, imac, hardware
He borrowed the money from **banks**.	banking, credit, investment, finance, citibank, currency, assets, loads, imf, hsbc
along the shores of lakes, **banks** of rivers	land, coast, river, waters, stream, inland, area, coasts, shoreline, shores, peninsula
Basalt is the commonest volcanic **rock**.	boulder, stone, rocks, sand, mud, limestone, volcanic, sedimentary, pelt, lava, basalt
Rock is the music of teenage rebellion.	band, pop, bands, song, rap, album, jazz. blues, singer, hip-pop, songs, guitar, musician

Table 2: Nearest neighbors of words given context. The embeddings from context words are first inferred with the Greedy strategy; nearest neighbors are computed by cosine similarity between word embeddings. Similar phenomena have been observed in earlier work (Neelakantan et al., 2014)

Neelakantan et al. (2014): *Efficient nonparametric estimation of multiple embeddings per word in vector space*

Li & Jurafsky (2015): *Do Multi-Sense Embeddings Improve Natural Language Understanding?*

Multisense Word Embeddings

- limitations:
  - need a way to label senses or cluster word tokens in training data (and for downstream tasks)
  - fragments training data, so more may be needed for estimating word embeddings
  - unlikely to get good clusters for rare word types
  - unable to handle new senses that only appear in test data
  - unclear if sense-specific embeddings are useful for downstream tasks

Do Multisense Embeddings Help on NLP Tasks?

- yes, on some tasks
- but when using powerful neural architectures, multisense embeddings may not be needed

We then test the performance of our model on part-of-speech tagging, named entity recognition, sentiment analysis, semantic relation identification and semantic relatedness, controlling for embedding dimensionality. We find that multi-sense embeddings do improve performance on some tasks (part-of-speech tagging, semantic relation identification, semantic relatedness) but not on others (named entity recognition, various forms of sentiment analysis).

- increasing dimensionality of (single-sense) embeddings achieves some benefit of multisense embeddings
  - high dimensionality also may make it easier for subsequent architectures to extract relevant sense based on context

Li & Jurafsky (2015): *Do Multi-Sense Embeddings Improve Natural Language Understanding?*