Introduction to NLP

CSE5321/CSEG321

Lecture 4. Word vector 2

Hwaran Lee (hwaranlee@sogang.ac.kr)

Materials are referenced from Stanford CS224n

Lecture Plan

Lecture 4: Word Vectors, Word Senses, and Neural Network Classifiers

1. Notice
2. Finish looking at word vectors and word2vec
3. Can we capture the essence of word meaning more effectively by counting?
4. Evaluating word vectors
5. Word senses
6. Review of classification and how neural nets differ

Key Goal: To be able to read word embeddings papers by the end of class

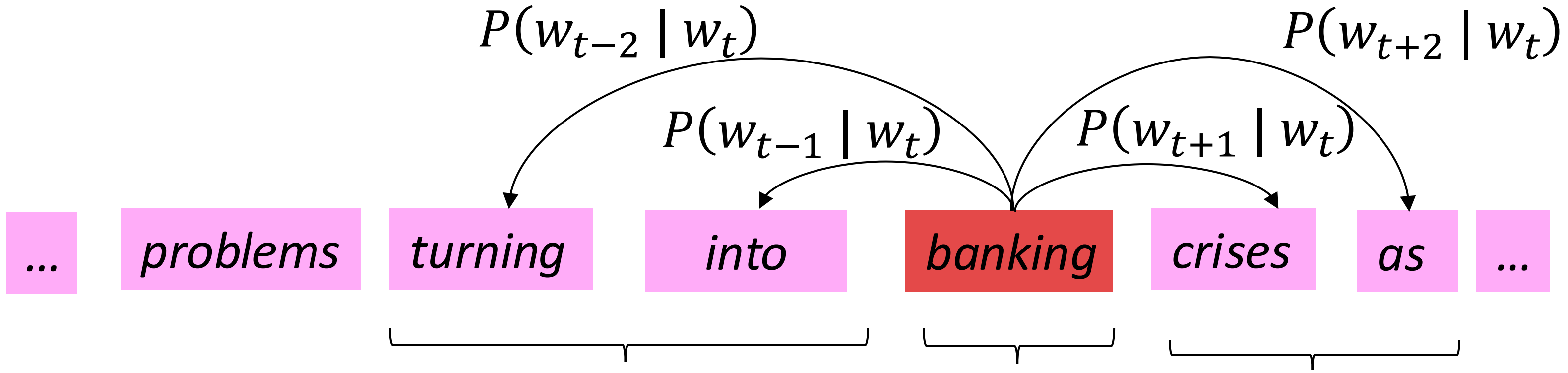
Notice

- Slido: #sogang-nlp2025w{#week-number}
    - Each week starts from Tuesday
- Homework #1 will be released Mar 18 at CyberCampus

Join at
slido.com
#sogang-nlp2025w2

2. Review: Main idea of word2vec

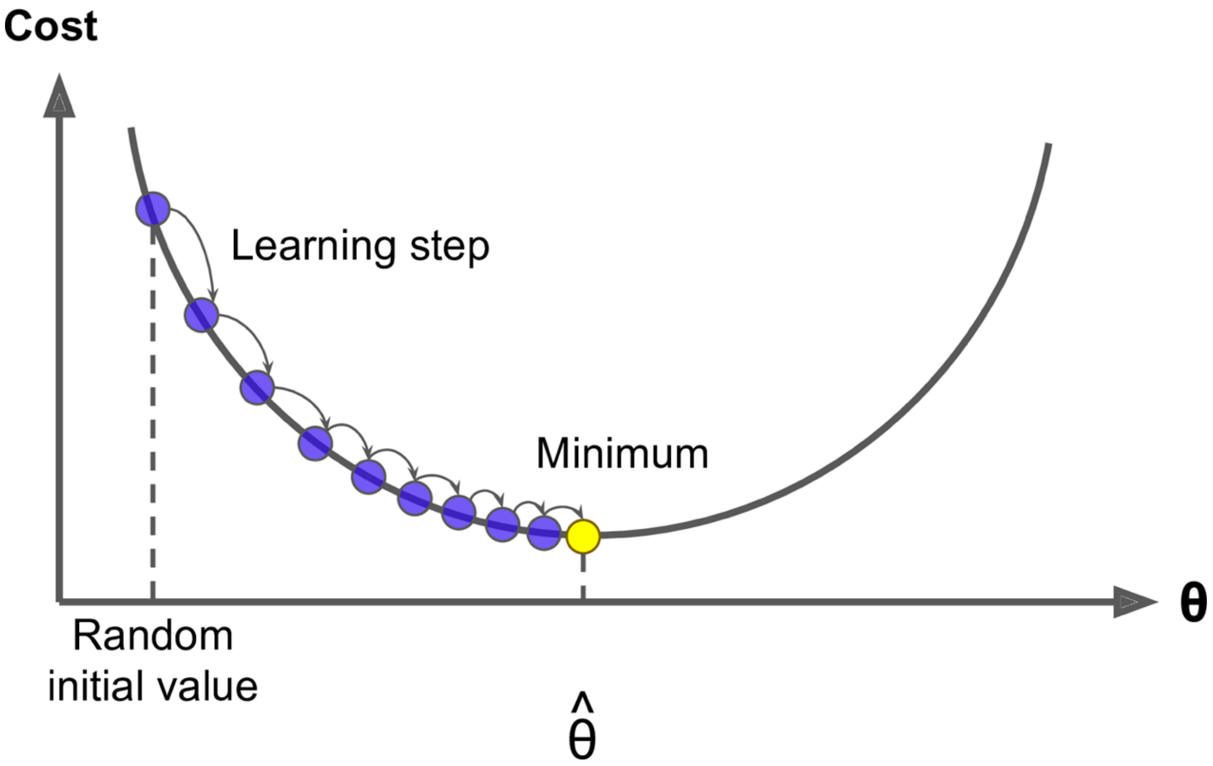
- Start with random word vectors
- Iterate through each word position in the whole corpus
- Try to predict surrounding words using word vectors: $P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$



- **Learning:** Update vectors so they can predict actual surrounding words better
- Doing no more than this, this algorithm learns word vectors that capture well word similarity and meaningful directions in a word space!

2. Optimization: Gradient Descent

- We have a cost function $J(\theta)$ we want to minimize
- **Gradient Descent** is an algorithm to minimize $J(\theta)$
- **Idea:** for current value of θ, calculate gradient of $J(\theta)$, then take small step in direction of negative gradient. Repeat.



Note: Our objectives may not be convex like this ☹

But life turns out to be okay ☺

Gradient Descent

- Update equation (in matrix notation):

$$\theta^{new} = \theta^{old} - \alpha \nabla_\theta J(\theta)$$

α = *step size* or *learning rate*

- Update equation (for single parameter):

$$\theta_j^{new} = \theta_j^{old} - \alpha \frac{\partial}{\partial \theta_j^{old}} J(\theta)$$

- Algorithm:

```
while True:
    theta_grad = evaluate_gradient(J,corpus,theta)
    theta = theta - alpha * theta_grad
```

Stochastic Gradient Descent

- **Problem**: $J(\theta)$ is a function of **all** windows in the corpus (potentially billions!)
  - So $\nabla_\theta J(\theta)$ is very expensive to compute
- You would wait a very long time before making a single update!

- **Very** bad idea for pretty much all neural nets!
- **Solution: Stochastic gradient descent (SGD)**
  - Repeatedly sample windows, and update after each one
- Algorithm:

```
while True:
    window = sample_window(corpus)
    theta_grad = evaluate_gradient(J,window,theta)
    theta = theta - alpha * theta_grad
```

Mini Batch Gradient Descent

Word2vec parameters … and computations

$$\begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix} \quad \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix} \qquad \begin{bmatrix} \bullet \\ \bullet \\ \bullet \\ \bullet \\ \bullet \\ \bullet \end{bmatrix} \quad \begin{bmatrix} \bullet \\ \bullet \\ \bullet \\ \bullet \\ \bullet \\ \bullet \end{bmatrix}$$

U outside

V center

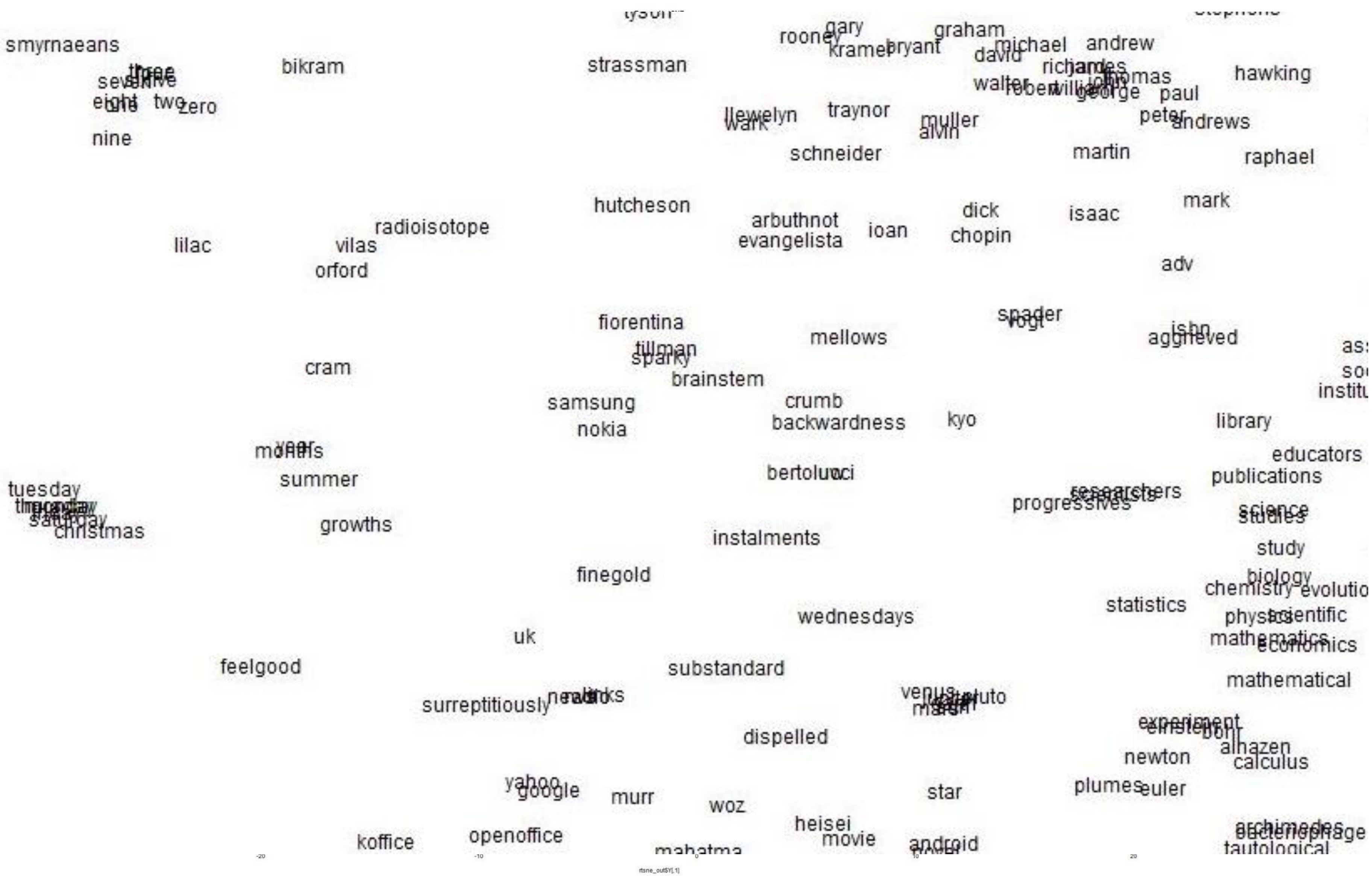
$U \cdot v_4{}^T$ dot product

$\mathrm{softmax}(U \cdot v_4{}^T)$ probabilities

"Bag of words" model! → The model makes the same predictions at each position

We want a model that gives a reasonably high probability estimate to *all* words that occur in the context (at all often)

Word2vec maximizes objective function by putting similar words nearby in space



Word2vec algorithm family (Mikolov et al. 2013): More details

Why two vectors? → Easier optimization. Average both at the end

- But can implement the algorithm with just one vector per word … and it helps a bit

Two model variants:

1. Skip-grams (SG)

   Predict context ("outside") words (position independent) given center word

2. Continuous Bag of Words (CBOW)

   Predict center word from (bag of) context words

We presented: **Skip-gram model**

Loss functions for training:

1. Naïve softmax (simple but expensive loss function, when many output classes)
2. More optimized variants like hierarchical softmax
3. Negative sampling

So far, we explained **naïve softmax**

The skip-gram model with negative sampling

- The normalization term is computationally expensive (when many output classes):

- $P(o|c) = \dfrac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$

A big sum over words

- Hence, standard word2vec implements the skip-gram model with **negative sampling**

- Main idea: train binary logistic regressions to differentiate a true pair (center word and a word in its context window) versus several “noise” pairs (the center word paired with a random word)

The skip-gram model with negative sampling ([Mikolov et al. 2013])

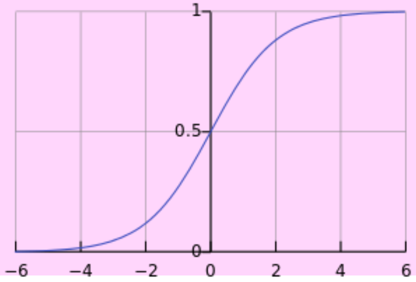
- We take *k* negative samples (using word probabilities)
- Maximize probability that real outside word appears;
  minimize probability that random words appear around center word
- Using notation consistent with this class, we minimize:

$$J_{neg-sample}(\boldsymbol{u}_o, \boldsymbol{v}_c, U) = -\log \sigma(\boldsymbol{u}_o^T \boldsymbol{v}_c) - \sum_{k \in \{K\ sampled\ indices\}} \log \sigma(-\boldsymbol{u}_k^T \boldsymbol{v}_c)$$

sigmoid rather than softmax

- The logistic/sigmoid function:
  (we'll become good friends soon)

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



- Sample with $P(w)=U(w)^{3/4}/Z$, the unigram distribution U(w) raised to the 3/4 power
  - The power makes less frequent words be sampled more often

Stochastic gradients with negative sampling [aside]

- We iteratively take gradients at each window for SGD
- In each window, we only have at most $2m + 1$ words plus $2km$ negative words with negative sampling, so $\nabla_\theta J_t(\theta)$ is very sparse!

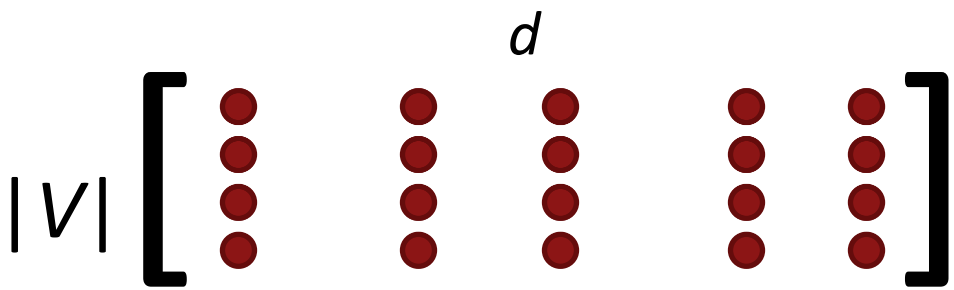
$$\nabla_\theta J_t(\theta) = \begin{bmatrix} 0 \\ \vdots \\ \nabla_{v_{like}} \\ \vdots \\ 0 \\ \nabla_{u_I} \\ \vdots \\ \nabla_{u_{learning}} \\ \vdots \end{bmatrix} \in \mathbb{R}^{2dV}$$

Stochastic gradients with with negative sampling [aside]

- We might only update the word vectors that actually appear!

- Solution: either you need sparse matrix update operations to only update certain rows of full embedding matrices *U* and *V*, or you need to keep around a hash for word vectors

Rows not columns in actual DL packages!



- If you have millions of word vectors and do distributed computing, it is important to not have to send gigantic updates around!

3. Why not capture co-occurrence counts directly?

There's something weird about iterating through the whole corpus (perhaps many times); why don't we just accumulate all the statistics of what words appear near each other?!?

Building a co-occurrence matrix X

- 2 options: windows vs. full document
- Window: Similar to word2vec, use window around each word → captures some syntactic and semantic information ("word space")
- Word-document co-occurrence matrix will give general topics (all sports terms will have similar entries) leading to "Latent Semantic Analysis" ("document space")

Example: Window based co-occurrence matrix

- Window length 1 (more common: 5–10)
- Symmetric (irrelevant whether left or right context)

- Example corpus:
  - I like deep learning
  - I like NLP
  - I enjoy flying

counts	I	like	enjoy	deep	learning	NLP	flying	.
I	0	2	1	0	0	0	0	0
like	2	0	0	1	0	1	0	0
enjoy	1	0	0	0	0	0	1	0
deep	0	1	0	0	1	0	0	0
learning	0	0	0	1	0	0	0	1
NLP	0	1	0	0	0	0	0	1
flying	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

Co-occurrence vectors

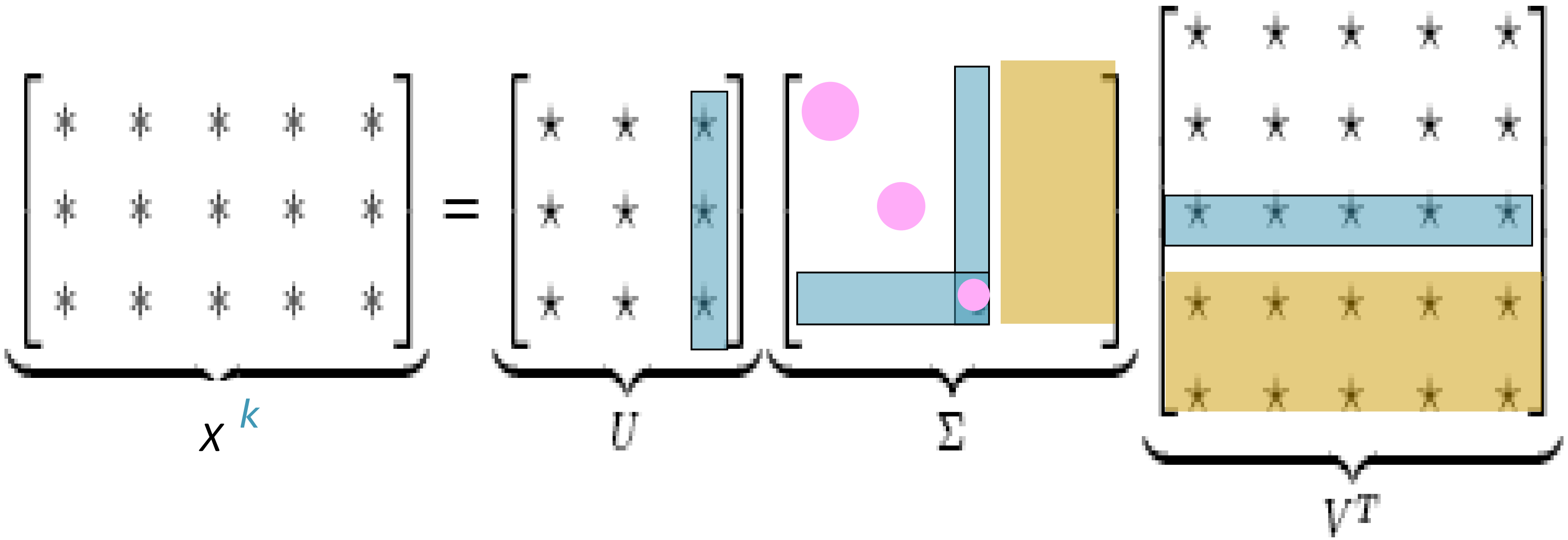
- Simple count co-occurrence vectors
  - Vectors increase in size with vocabulary
  - Very high dimensional: require a lot of storage (though sparse)
  - Subsequent classification models have sparsity issues → Models are less robust

- Low-dimensional vectors
  - Idea: store "most" of the important information in a fixed, small number of dimensions: a dense vector
  - Usually 25–1000 dimensions, similar to word2vec
  - How to reduce the dimensionality?

Classic Method: Dimensionality Reduction on X (HW1)

Singular Value Decomposition of co-occurrence matrix X

Factorizes X into $U\Sigma V^{T}$, where U and V are orthonormal (unit vectors and orthogonal)



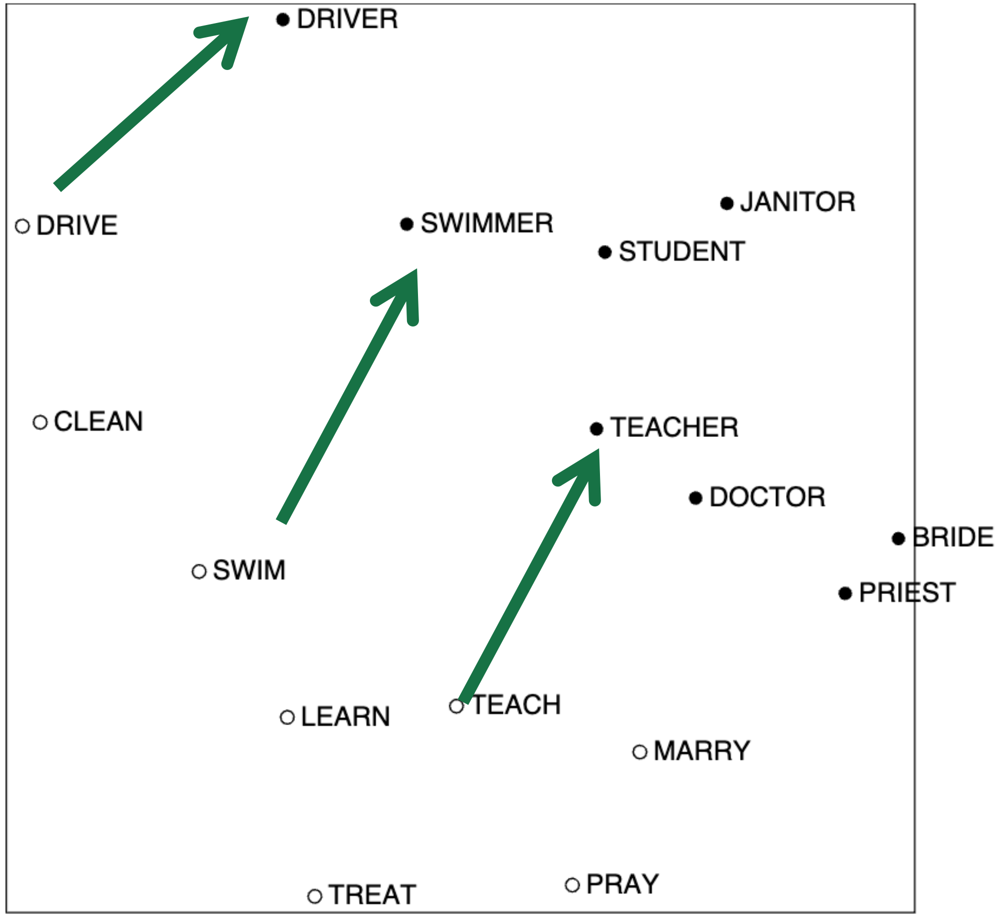
Retain only k singular values, in order to generalize.
$\hat{X}$ is the best rank k approximation to X , in terms of least squares.
Classic linear algebra result. Expensive to compute for large matrices.

Hacks to X

- Running an SVD on raw counts doesn't work well!!!

- Scaling the counts in the cells can help ***a lot***
  - Problem: function words (*the, he, has*) are too frequent → syntax has too much impact. Some fixes:
    - log the frequencies
    - min(X, t), with $t \approx 100$
    - Ignore the function words
- Ramped windows that count closer words more than further away words
- Use correlations instead of counts, then set negative values to 0
- Etc.

Interesting semantic patterns emerge in the scaled vectors



COALS model from
Rohde et al. ms., 2005. An Improved Model of Semantic Similarity Based on Lexical Co-Occurrence

GloVe [Pennington, Socher, and Manning, EMNLP 2014]: Encoding meaning components in vector differences

Q: How can we capture ratios of co-occurrence probabilities as linear meaning components in a word vector space?

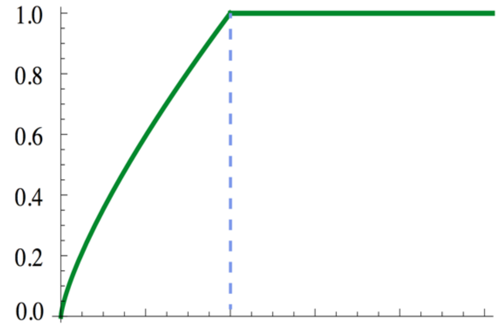
GloVe [Pennington, Socher, and Manning, EMNLP 2014]: **Encoding meaning components in vector differences**

Q: How can we capture ratios of co-occurrence probabilities as linear meaning components in a word vector space?

A: Log-bilinear model: $$w_i \cdot w_j = \log P(i|j)$$

with vector differences $$w_x \cdot (w_a - w_b) = \log \frac{P(x|a)}{P(x|b)}$$

Loss: $$J = \sum_{i,j=1}^{V} f\left(X_{ij}\right)\left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij}\right)^2$$

$f \sim$ 

- Fast training
- Scalable to huge corpora

4. How to evaluate word vectors?

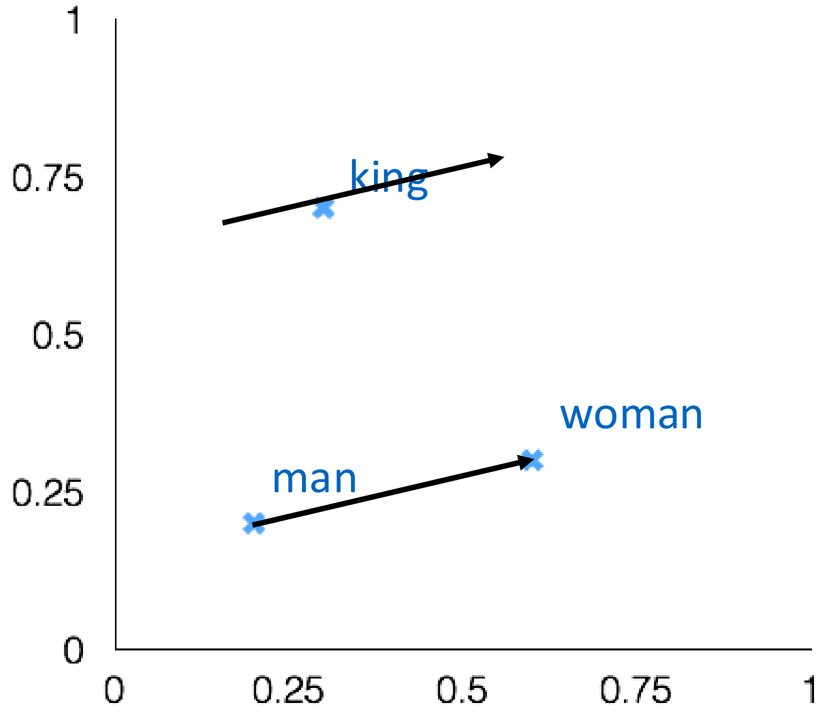
- Related to general evaluation in NLP: Intrinsic vs. extrinsic
- Intrinsic:
  - Evaluation on a specific/intermediate subtask
  - Fast to compute
  - Helps to understand that system
  - Not clear if it’s helpful unless correlation to real task is established
- Extrinsic:
  - Evaluation on a real task
  - Can take a long time to compute accuracy
  - Unclear if the subsystem is the problem or its interaction or other subsystems
  - If replacing exactly one subsystem with another improves accuracy → Winning!

Intrinsic word vector evaluation

- Word Vector Analogies

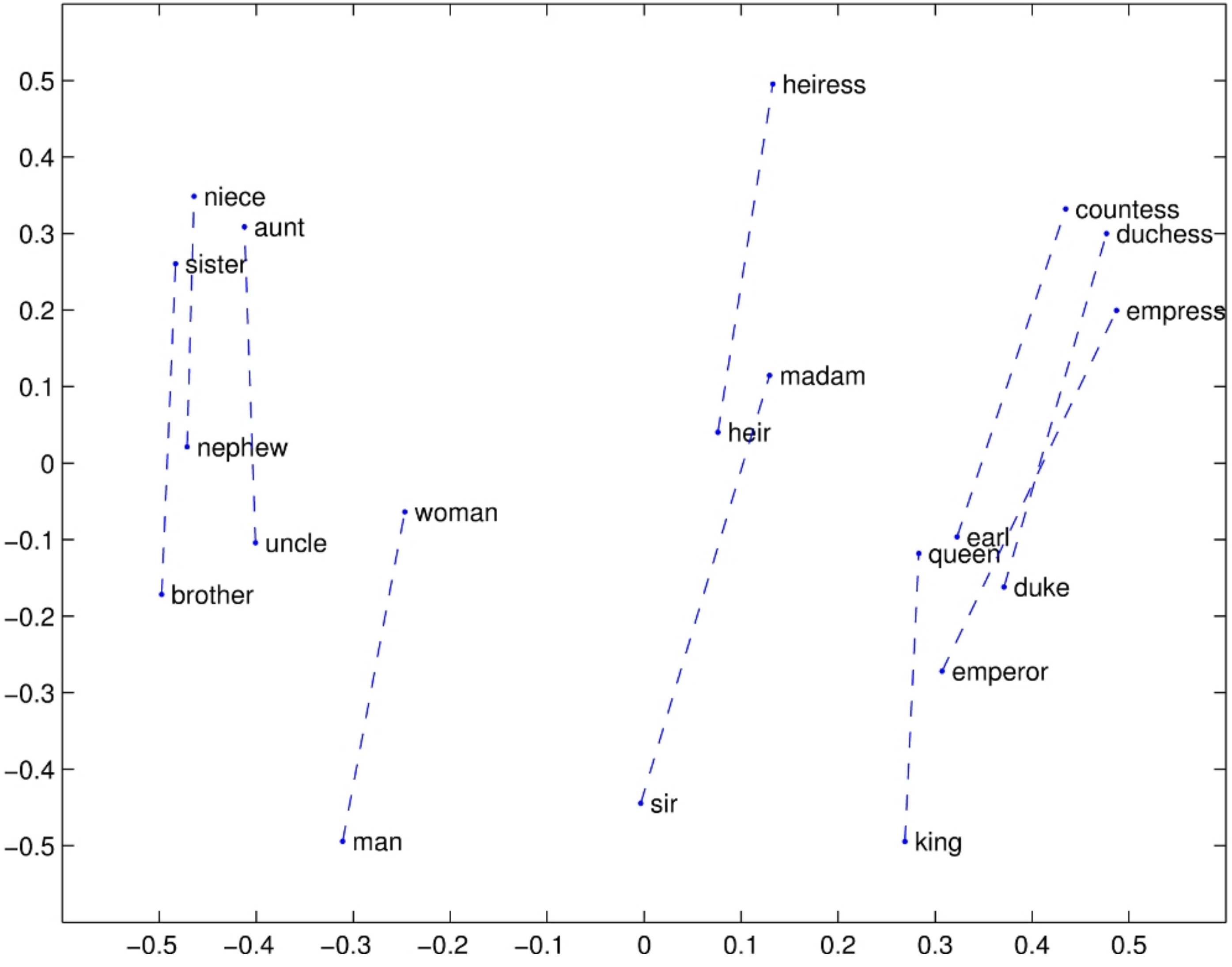
$$\boxed{a{:}b :: c{:}?} \longrightarrow \boxed{d = \arg\max_i \frac{(x_b - x_a + x_c)^T x_i}{||x_b - x_a + x_c||}}$$

man:woman :: king:?



- Evaluate word vectors by how well their cosine distance after addition captures intuitive semantic and syntactic analogy questions
- Discarding the input words from the search (!)
- Problem: What if the information is there but not linear?

GloVe Visualization



Meaning similarity: Another intrinsic word vector evaluation

- Word vector distances and their correlation with human judgments
- Example dataset: WordSim353
  http://www.cs.technion.ac.il/~gabr/resources/data/wordsim353/

Word 1	Word 2	Human (mean)
tiger	cat	7.35
tiger	tiger	10
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

Correlation evaluation

- Word vector distances and their correlation with human judgments

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	<u>72.7</u>	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	<u>58.1</u>	37.2
GloVe	6B	<u>65.8</u>	<u>72.7</u>	<u>77.8</u>	53.9	<u>38.1</u>
SVD-L	42B	74.0	76.4	74.1	58.3	39.9
GloVe	42B	<u>**75.9**</u>	<u>**83.6**</u>	<u>**82.9**</u>	<u>**59.6**</u>	<u>**47.8**</u>
CBOW*	100B	68.4	79.6	75.4	59.4	45.5

Extrinsic word vector evaluation

- One example where good word vectors should help directly: **named entity recognition**: identifying references to a person, organization or location: Chris Manning lives in Palo Alto.

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**

5. Word senses and word sense ambiguity

- Most words have lots of meanings!
  - Especially common words
  - Especially words that have existed for a long time

- Example: **pike**

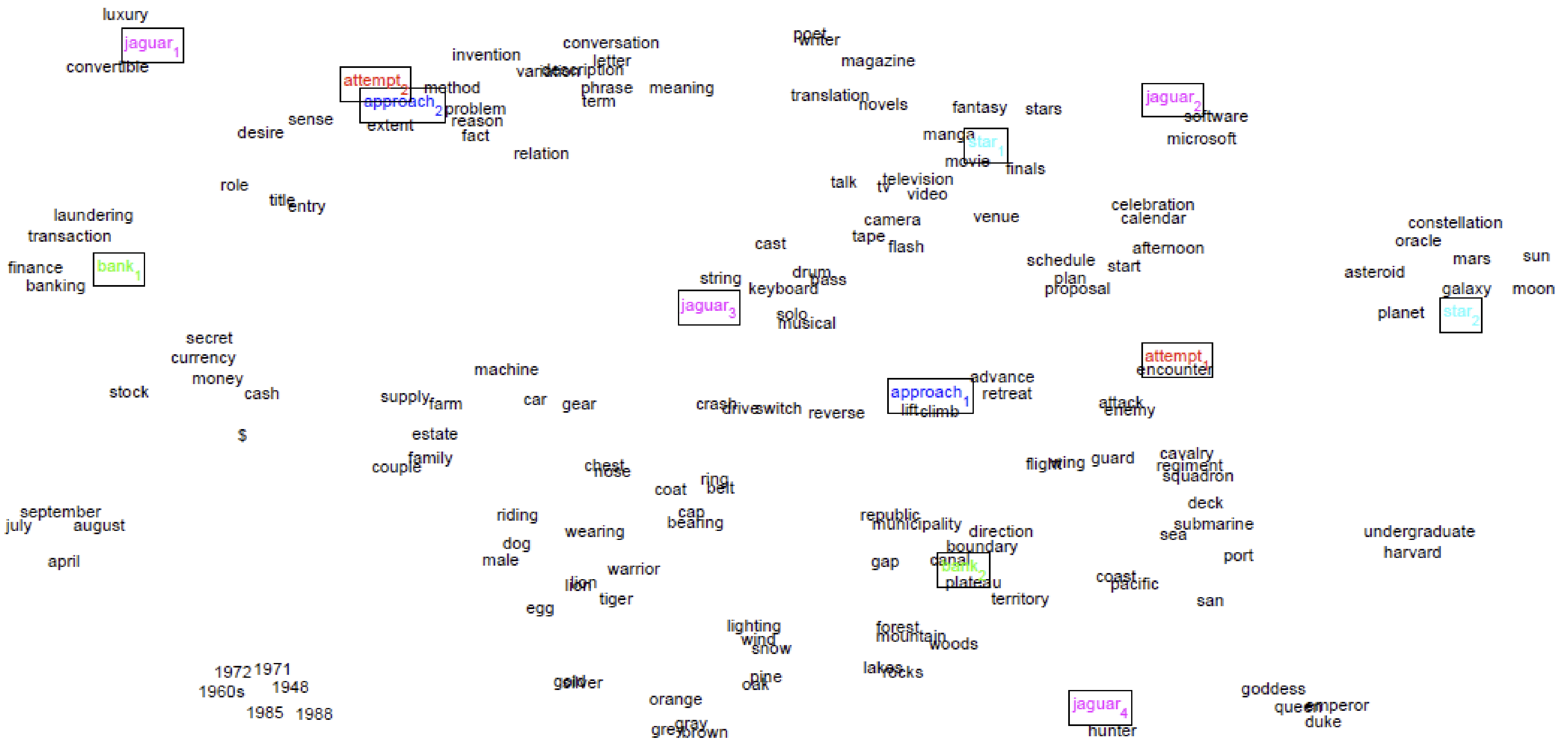
- Does one vector capture all these meanings or do we have a mess?

pike

- A sharp point or staff
- A type of elongated fish
- A railroad line or system
- A type of road
- The future (coming down the pike)
- A type of body position (as in diving)
- To kill or pierce with a pike
- To make one’s way (pike along)
- In Australian English, pike means to pull out from doing something: *I reckon he could have climbed that cliff, but he piked!*

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

- Idea: Cluster word windows around words, retrain with each word assigned to multiple different clusters bank$_1$, bank$_2$, etc.



Linear Algebraic Structure of Word Senses, with Applications to Polysemy (Arora, …, Ma, …, TACL 2018)

- Different senses of a word reside in a linear superposition (weighted sum) in standard word embeddings like word2vec
- $v_{\text{pike}} = \alpha_1 v_{\text{pike}_1} + \alpha_2 v_{\text{pike}_2} + \alpha_3 v_{\text{pike}_3}$
- Where $\alpha_1 = \frac{f_1}{f_1+f_2+f_3}$, etc., for frequency f
- Surprising result:
  - Because of ideas from *sparse coding* you can actually separate out the senses (providing they are relatively common)!

tie				
trousers	season	scoreline	wires	operatic
blouse	teams	goalless	cables	soprano
waistcoat	winning	equaliser	wiring	mezzo
skirt	league	clinching	electrical	contralto
sleeved	finished	scoreless	wire	baritone
pants	championship	replay	cable	coloratura

6. Deep Learning Classification: Named Entity Recognition (NER)

- The task: find and classify names in text, by labeling word tokens, for example:

Last night , Paris Hilton wowed in a sequin gown .
PER PER

Samuel Quinn was arrested in the Hilton Hotel in Paris in April 1989 .
PER PER LOC LOC LOC DATE DATE

- Possible uses:
  - Tracking mentions of particular entities in documents
  - For question answering, answers are usually named entities
  - Relating sentiment analysis to the entity under discussion
- Often followed by Entity Linking/Canonicalization into a Knowledge Base such as Wikidata

Simple NER: Window classification using binary logistic classifier

- **Idea:** classify each word in its context window of neighboring words
- Train logistic classifier on hand-labeled data to classify center word {yes/no} for each class based on a concatenation of word vectors in a window
  - Really, we usually use multi-class softmax, but we're trying to keep it simple ☺
- **Example:** Classify "Paris" as +/– location in context of sentence with window length 2:

the museums in Paris are amazing to see .

$$X_{window} = [\ x_{museums} \quad x_{in} \quad x_{Paris} \quad x_{are} \quad x_{amazing}\ ]^T$$

- Resulting vector $x_{window} = \boxed{x \in R^{5d}}$
- To classify all words: run classifier for each class on the vector centered on each word in the sentence

Classification review and notation

- Supervised learning: we have a training dataset consisting of samples

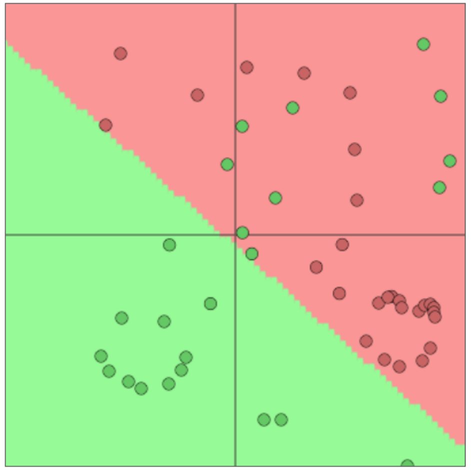
$$\{x_i,y_i\}^N_{i=1}$$

- x_i are inputs, e.g., words (indices or vectors!), sentences, documents, etc.
  - Dimension d

- y_i are labels (one of C classes) we try to predict, for example:
  - classes: sentiment (+/–), named entities, buy/sell decision
  - other words
  - later: multi-word sequences

Neural classification

- Typical ML/stats softmax classifier: $p(y|x) = \frac{\exp(W_y.x)}{\sum_{c=1}^{C}\exp(W_c.x)}$
- Learned parameters θ are just elements of *W* (not input representation *x*, which has sparse symbolic features)
- Classifier gives linear decision boundary, which can be limiting



- A **neural network classifier** differs in that:
  - We learn **both *W* and (distributed!)** representations for words
  - The word vectors *x* re-represent one-hot vectors, moving them around in an intermediate layer vector space, for easy classification with a (linear) softmax classifier
    - Conceptually, we have an embedding layer: *x* = *Le*
  - We use deep networks—more layers—that let us re-represent and compose our data multiple times, giving a non-linear classifier

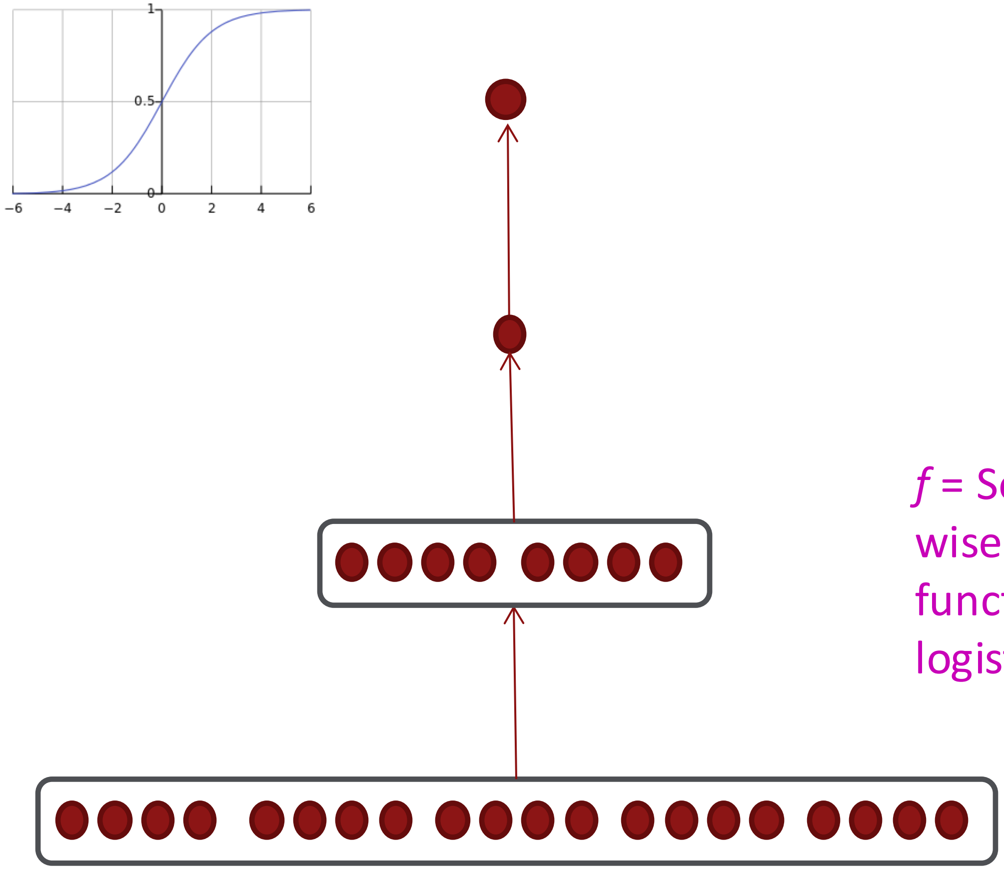
But typically, it is linear relative to the pre-final layer representation

NER: Binary classification for center word being location

- We do supervised training and want high score if it's a location

$$J_t(\theta) = \sigma(s) = \frac{1}{1+e^{-s}}$$

predicted model probability of class



$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

$$\boldsymbol{x} \quad \text{(input)}$$

$x = [\ x_{\text{museums}} \quad x_{\text{in}} \quad x_{\text{Paris}} \quad x_{\text{are}} \quad x_{\text{amazing}}\ ]$

f = Some element-wise non-linear function, e.g., logistic, tanh, ReLU

Training with "cross entropy loss" – you use this in PyTorch!

- Until now, our objective was stated as to maximize the probability of the correct class *y* or equivalently we can minimize the negative log probability of that class

- Now restated in terms of cross entropy, a concept from information theory
- Let the true probability distribution be *p; l*et our computed model probability be *q*
- The cross entropy is:

$$H(p, q) = -\sum_{c=1}^{C} p(c) \log q(c)$$

- Assuming a ground truth (or true or gold or target) probability distribution that is 1 at the right class and 0 everywhere else, *p* = [0, …, 0, 1, 0, …, 0], then:
- **Because of one-hot *p*, the only term left is the negative log probability of the true class y_i:** $-\log p(y_i|x_i)$

Cross entropy can be used in other ways with a more interesting *p*, but for now just know that you'll want to use it as the loss in PyTorch