

Natural Language Processing with Deep Learning

CS224N/Ling284



Christopher Manning

Lecture 2: Word Vectors, Word Senses, and Neural Classifiers

Lecture Plan

Lecture 2: Word Vectors, Word Senses, and Neural Network Classifiers

1. Course organization (2 mins)
2. Finish looking at word vectors and word2vec (10 mins)
3. Optimization basics (8 mins)
4. Can we capture the essence of word meaning more effectively by counting? (8m)
5. The GloVe model of word vectors (8 min)
6. Evaluating word vectors (12 mins)
7. Word senses (6 mins)
8. Review of classification and how neural nets differ (8 mins)
9. Introducing neural networks (14 mins)

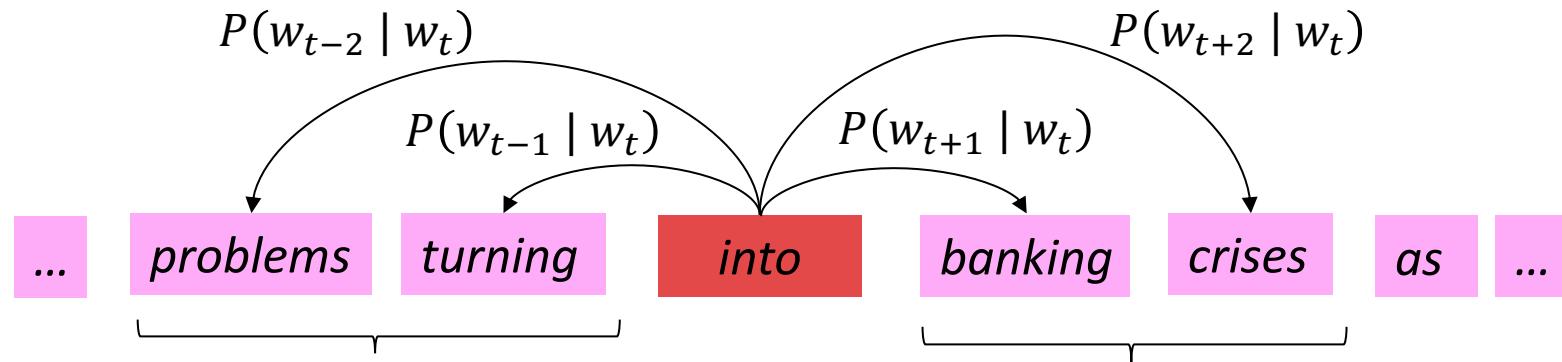
Key Goal: To be able to read word embeddings papers by the end of class

1. Course Organization

- **Come to office hours/help sessions!**
 - Come to discuss final project ideas as well as the assignments
 - Try to come early, often and off-cycle
- **TA office hours: 3-hour block on Mon, Tue, Wed, Thu, & Sat, with multiple TAs**
 - Just show up on **Nooks** and go to the Welcome room!
 - Our friendly course staff will be on hand to assist you
- **Chris's office hours:**
 - Mon 2–4pm. Book slot on **Calendly**. You can still come along this coming Monday!
- **Week 10:**
 - We lose a week this year with no exam week, so we need to do final projects sooner
 - No lectures in week 10; projects due Tue in week 10; extra project help sessions

2. Review: Main idea of word2vec

- Start with random word vectors
- Iterate through each word 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 wordspace!



Word2vec parameters and computations

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

U

outside

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

V

center

$$\begin{bmatrix} \bullet \\ \bullet \\ \bullet \\ \bullet \\ \bullet \end{bmatrix}$$

$U \cdot v_4^T$

dot product

$$\begin{bmatrix} \bullet \\ \bullet \\ \bullet \\ \bullet \\ \bullet \end{bmatrix}$$

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

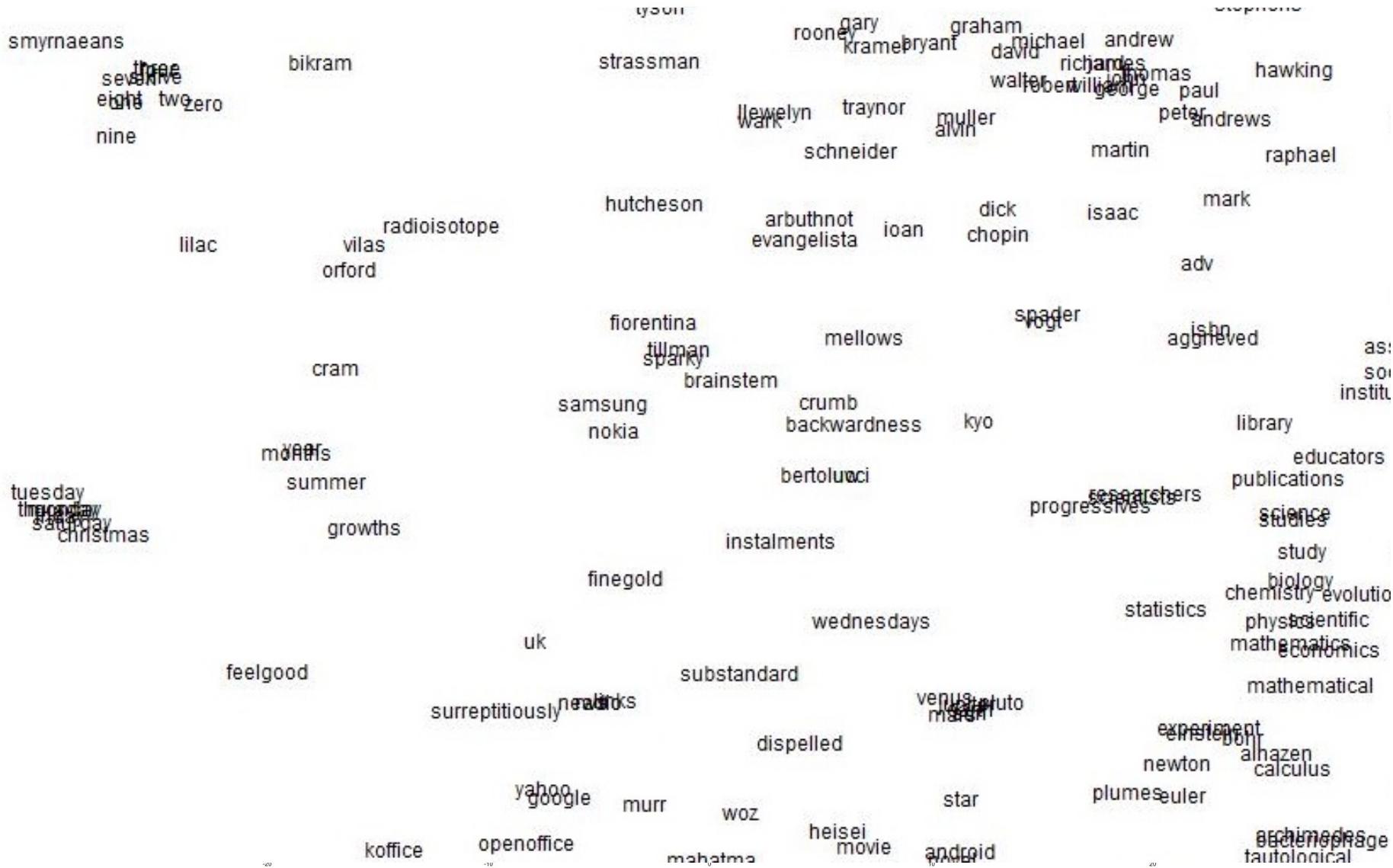
probabilities

“Bag of words” model!

The model makes the same predictions at each position

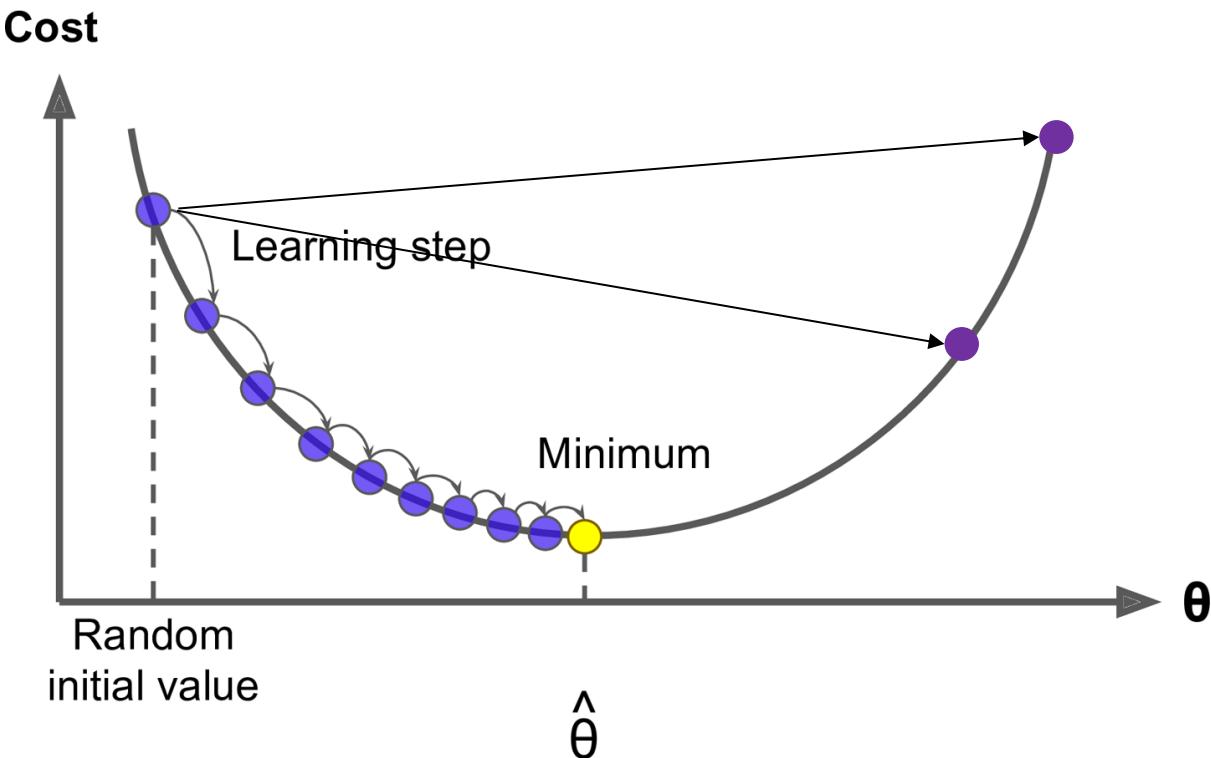
We want a model that gives a reasonably high probability estimate to *all* words that occur in the context (at all often)

Word2vec maximizes objective function by putting similar words nearby in space



3. Optimization: Gradient Descent

- To learn good word vectors: We have a cost function $J(\theta)$ we want to minimize
- **Gradient Descent** is an algorithm to minimize $J(\theta)$ by changing θ
- **Idea:** from current value of θ , calculate gradient of $J(\theta)$, then take **small step in the 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 a 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 (often, 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, or each small batch
- Algorithm:

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

Stochastic gradients with word vectors! [Aside]

- Iteratively take gradients at each such window for SGD
- But in each window, we only have at most $2m + 1$ words, 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 word vectors!

- 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!

$$|V| \begin{bmatrix} \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}^d$$

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

2b. Word2vec algorithm family: 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

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**

Additional efficiency in training:

1. Negative sampling

So far: Focus on **naïve softmax** (simpler, but expensive, training method)

The skip-gram model with negative sampling (HW2)

- The normalization term is computationally expensive
- $$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$
- Hence, in standard word2vec and HW2 you implement the skip-gram model with **negative sampling**
- Main idea: train binary logistic regressions for a true pair (center word and a word in its context window) versus several noise pairs (the center word paired with a random word)

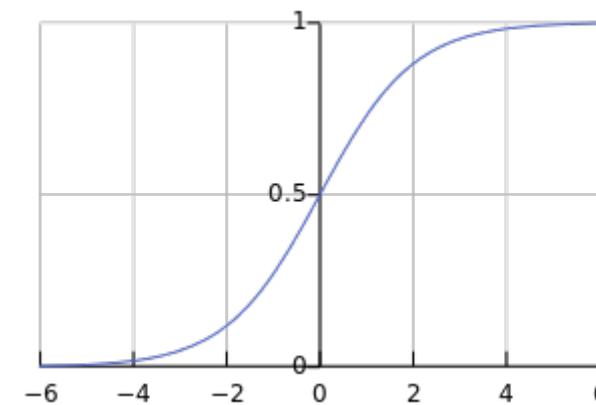
The skip-gram model with negative sampling (HW2)

- From paper: “Distributed Representations of Words and Phrases and their Compositionality” (Mikolov et al. 2013)
- Overall objective function (they maximize):

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

$$J_t(\theta) = \log \sigma(u_o^T v_c) + \sum_{i=1}^k \mathbb{E}_{j \sim P(w)} [\log \sigma(-u_j^T v_c)]$$

- The logistic/sigmoid function: $\sigma(x) = \frac{1}{1+e^{-x}}$
(we'll become good friends soon)
- We maximize the probability
of two words co-occurring in first log
and minimize probability of noise words



The skip-gram model with negative sampling (HW2)

- Notation more similar to class and HW2:

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

- We take k negative samples (using word probabilities)
- Maximize probability that real outside word appears, minimize probability that random words appear around center word
- Sample with $P(w)=U(w)^{3/4}/Z$, the unigram distribution $U(w)$ raised to the $3/4$ power (We provide this function in the starter code).
- The power makes less frequent words be sampled more often

4. Why not capture co-occurrence counts directly?

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-document co-occurrence matrix will give general topics (all sports terms will have similar entries) leading to “Latent Semantic Analysis”

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

$$\underbrace{\begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}}_{X^k} = \underbrace{\begin{bmatrix} * & * \\ * & * \\ * & * \end{bmatrix}}_U \underbrace{\begin{bmatrix} \bullet & & \\ & \bullet & \\ & & \bullet \end{bmatrix}}_{\Sigma} \underbrace{\begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}}_{V^T}$$

Retain only k singular values, in order to generalize.

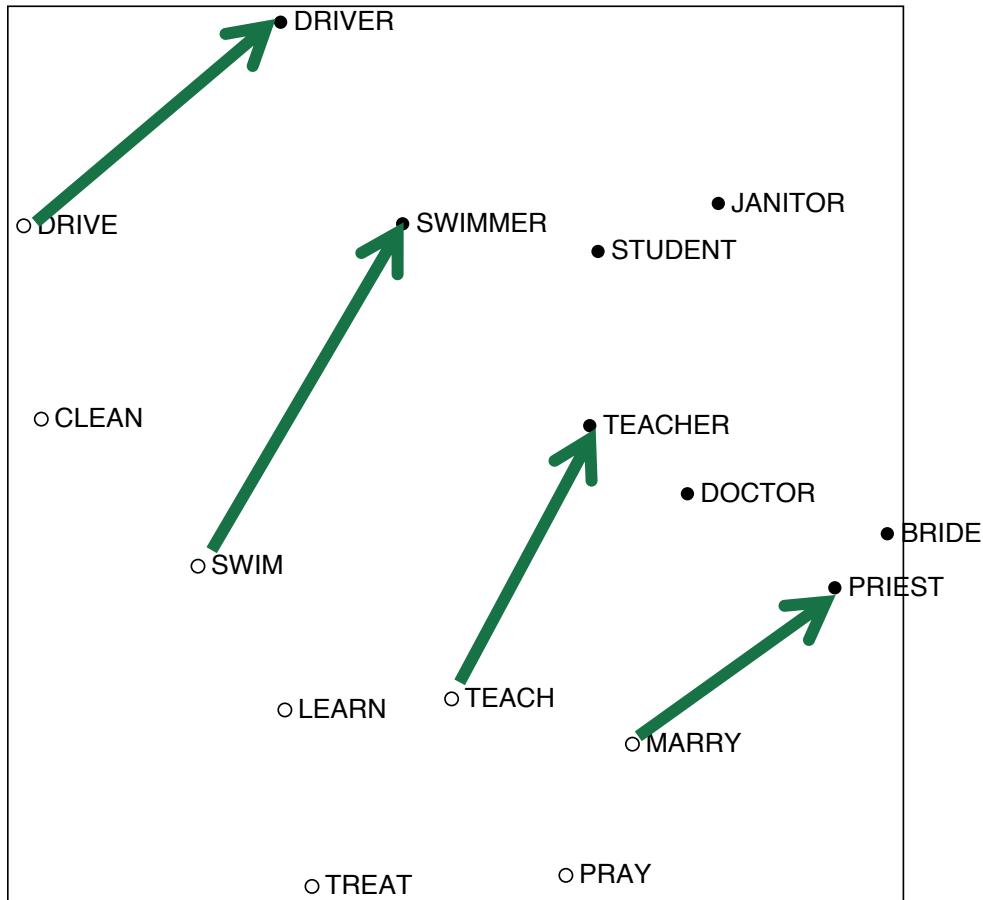
\hat{X} is the best rank k approximation to X , in terms of least squares.

Classic linear algebra result. Expensive to compute for large matrices.

Hacks to X (several used in Rohde et al. 2005 in COALS)

- 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 Pearson 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

5. Towards GloVe: Count based vs. direct prediction

- LSA, HAL (Lund & Burgess),
- COALS, Hellinger-PCA (Rohde et al, Lebret & Collobert)

- Fast training
- Efficient usage of statistics
- Primarily used to capture word similarity
- Disproportionate importance given to large counts

- Skip-gram/CBOW (Mikolov et al)
- NNLM, HLBL, RNN (Bengio et al; Collobert & Weston; Huang et al; Mnih & Hinton)

- Scales with corpus size
- Inefficient usage of statistics
- Generate improved performance on other tasks
- Can capture complex patterns beyond word similarity

Encoding meaning components in vector differences

[Pennington, Socher, and Manning, EMNLP 2014]

Crucial insight: Ratios of co-occurrence probabilities can encode meaning components

	$x = \text{solid}$	$x = \text{gas}$	$x = \text{water}$	$x = \text{random}$
$P(x \text{ice})$	large	small	large	small
$P(x \text{steam})$	small	large	large	small
$\frac{P(x \text{ice})}{P(x \text{steam})}$	large	small	~1	~1

Encoding meaning in vector differences

[Pennington, Socher, and Manning, EMNLP 2014]

Crucial insight: Ratios of co-occurrence probabilities can encode meaning components

	$x = \text{solid}$	$x = \text{gas}$	$x = \text{water}$	$x = \text{fashion}$
$P(x \text{ice})$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(x \text{steam})$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$\frac{P(x \text{ice})}{P(x \text{steam})}$	8.9	8.5×10^{-2}	1.36	0.96

Encoding meaning 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

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

Combining the best of both worlds

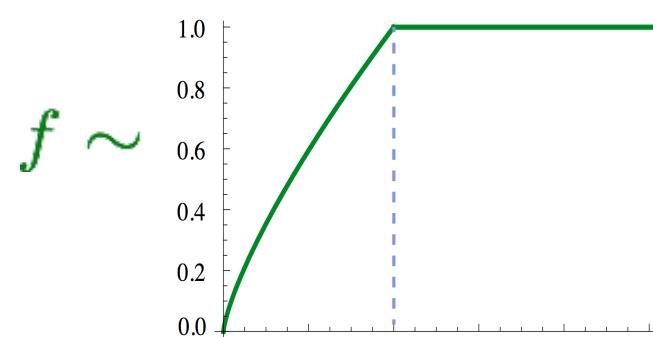
GloVe [Pennington, Socher, and Manning, EMNLP 2014]

$$w_i \cdot w_j = \log P(i|j)$$



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

- Fast training
- Scalable to huge corpora
- Good performance even with small corpus and small vectors



GloVe results

Nearest words to
[frog](#):

1. frogs
2. toad
3. litoria
4. leptodactylidae
5. rana
6. lizard
7. eleutherodactylus



[litoria](#)



[rana](#)



[leptodactylidae](#)



[eleutherodactylus](#)

6. 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 really 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

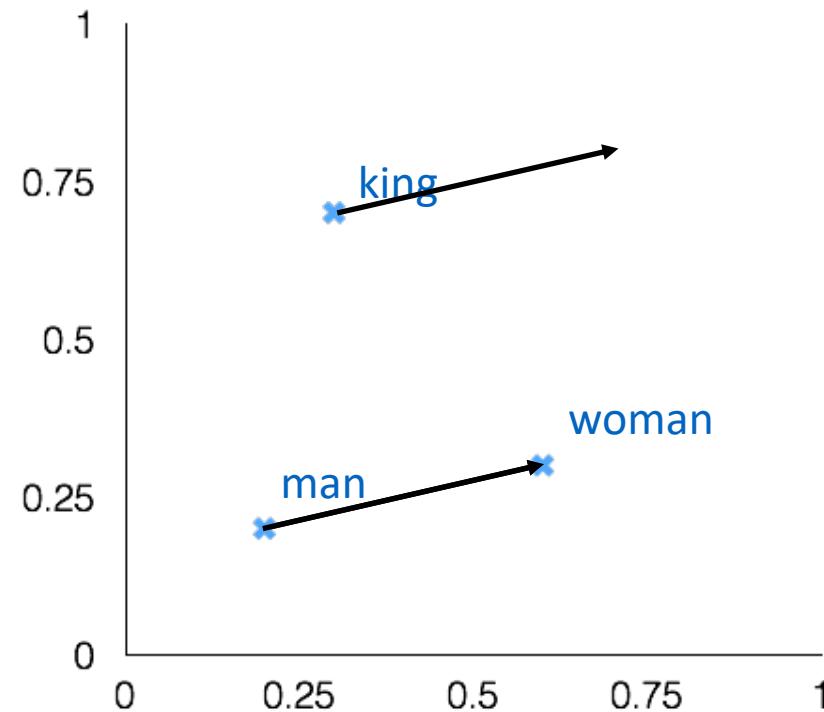
$$a:b :: c:?$$



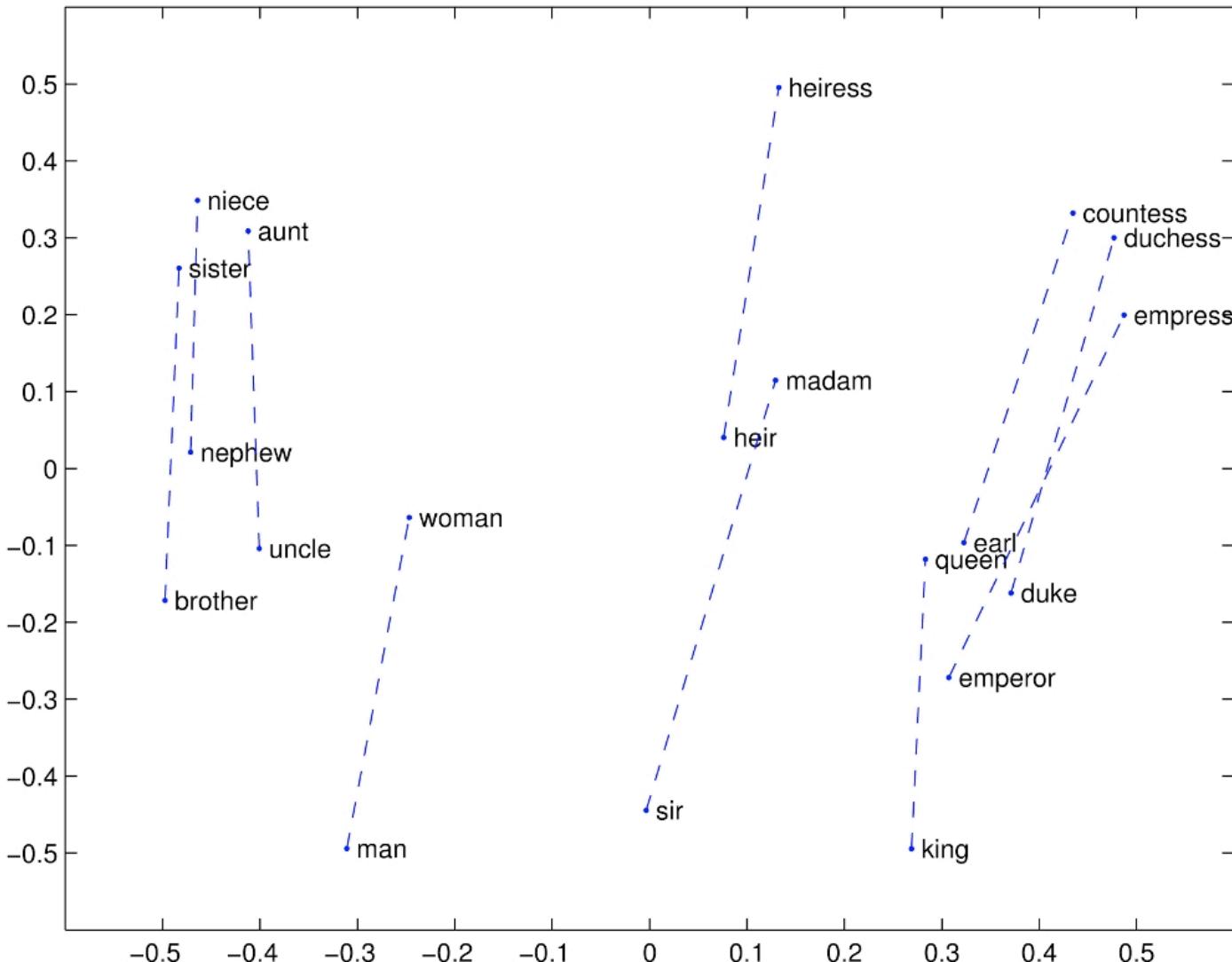
man:woman :: king:?

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

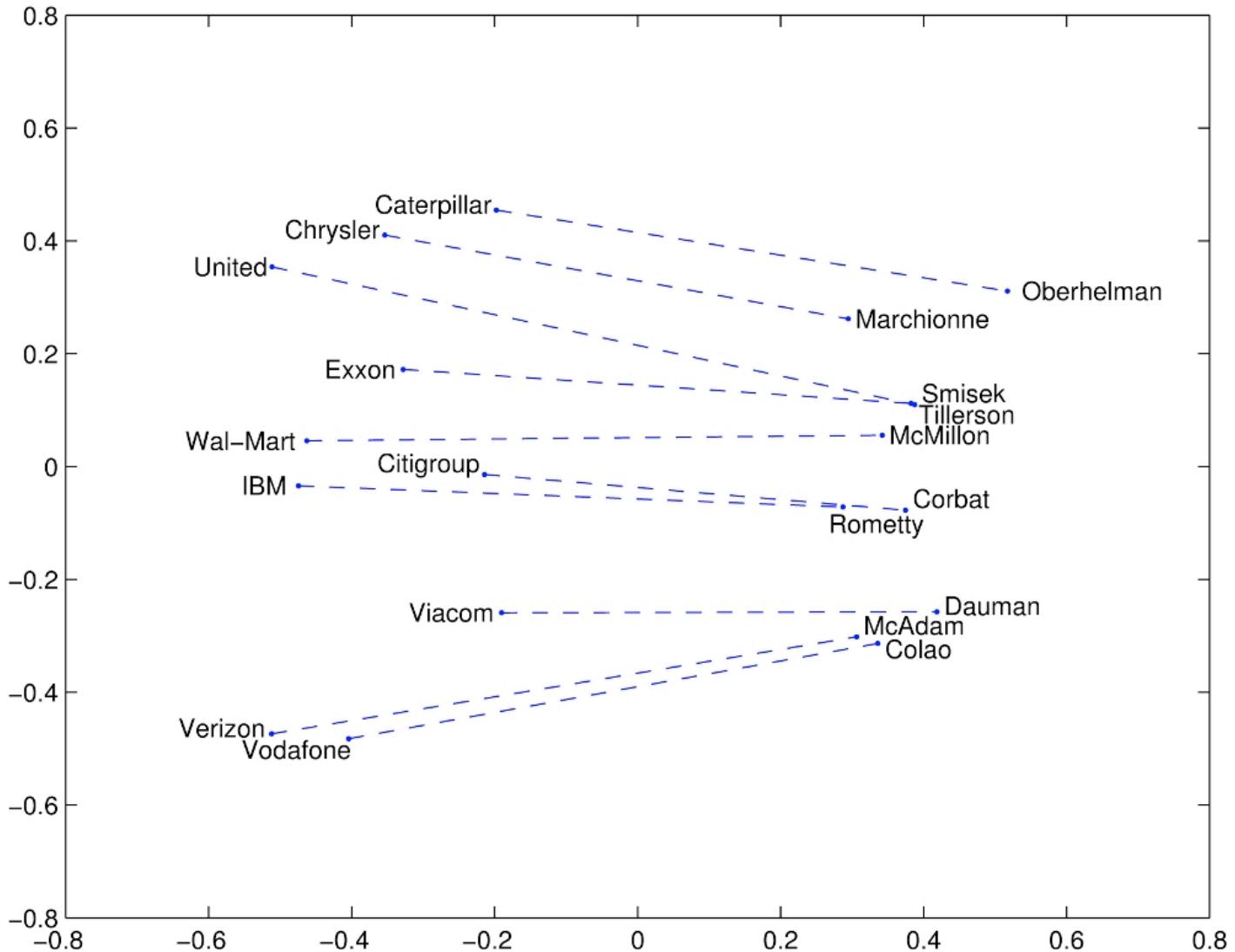
- Evaluate word vectors by how well their cosine distance after addition captures intuitive semantic and syntactic analogy questions
- Discarding the input words from the search!
- Problem: What if the information is there but not linear?



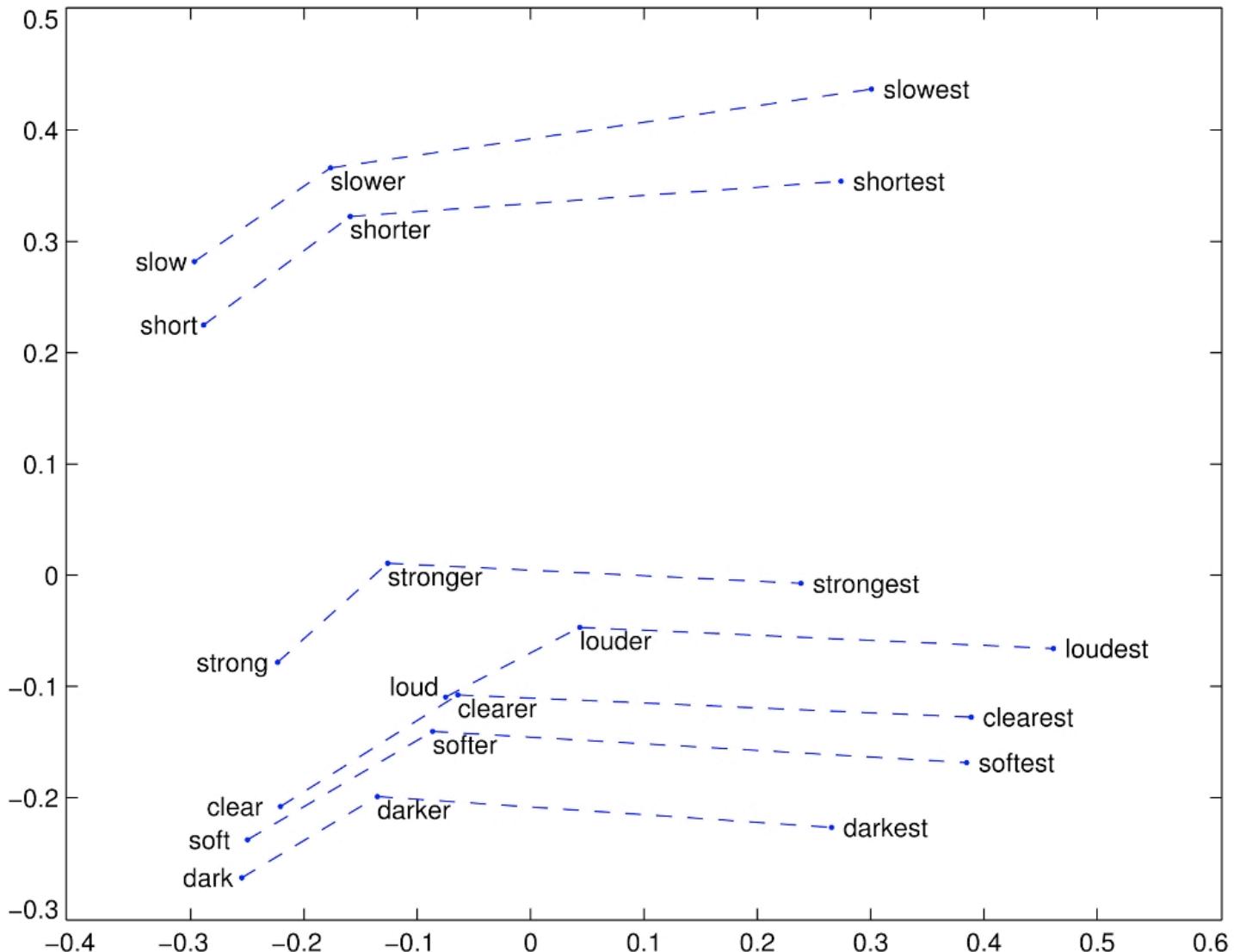
Glove Visualizations



Glove Visualizations: Company - CEO



Glove Visualizations: Comparatives and Superlatives



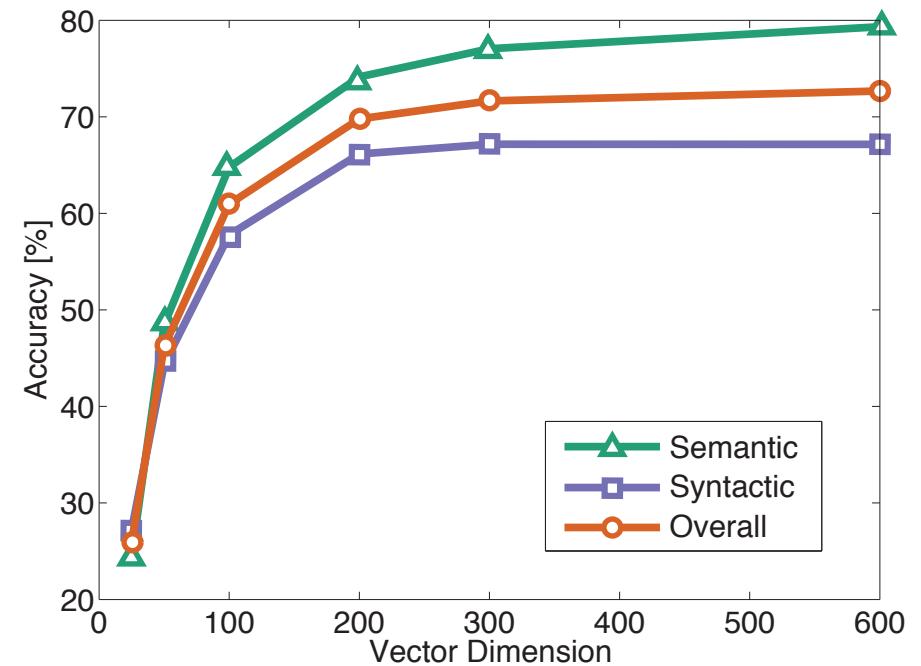
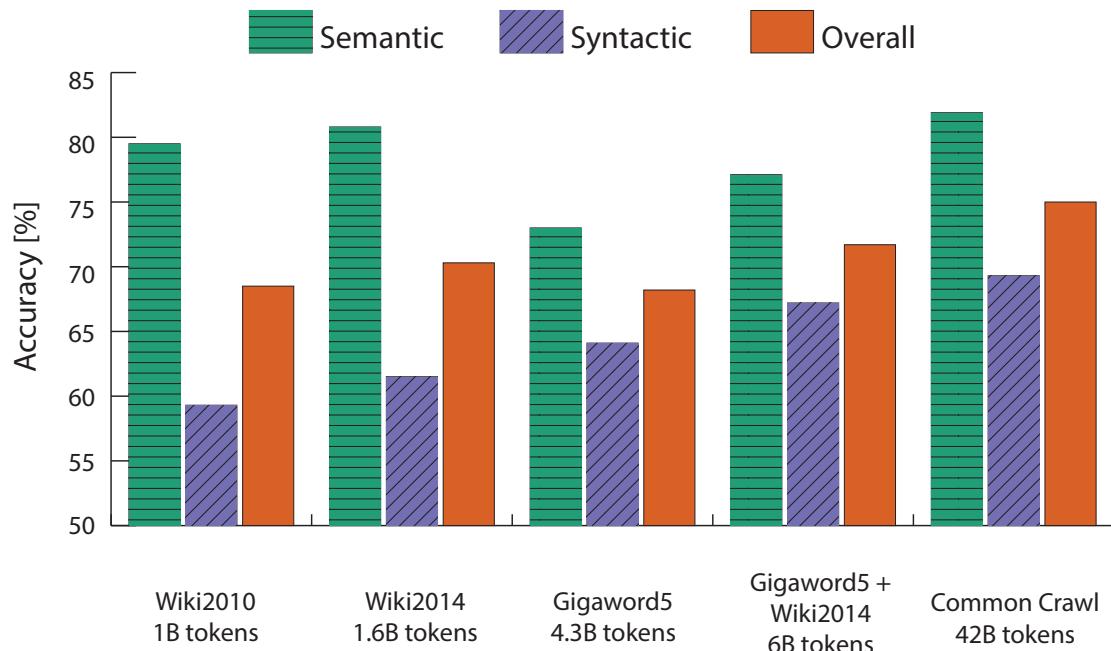
Analogy evaluation and hyperparameters

Glove word vectors evaluation

Model	Dim.	Size	Sem.	Syn.	Tot.
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	<u>67.4</u>	65.7
SG [†]	300	6B	73.0	66.0	69.1
GloVe	300	6B	<u>77.4</u>	67.0	<u>71.7</u>

Analogy evaluation and hyperparameters

- More data helps
- Wikipedia is better than news text!
- Dimensionality
- Good dimension is ~ 300



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	<u>65.6</u>	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	75.9	83.6	82.9	59.6	47.8
CBOW*	100B	68.4	79.6	75.4	59.4	45.5

- Some ideas from Glove paper have been shown to improve skip-gram (SG) model also (e.g., average both vectors)

Extrinsic word vector evaluation

- Extrinsic evaluation of word vectors: All subsequent NLP tasks in this class. More examples soon.
- One example where good word vectors should help directly: **named entity recognition**: identifying references to a person, organization or location

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

7. Word senses and word sense ambiguity

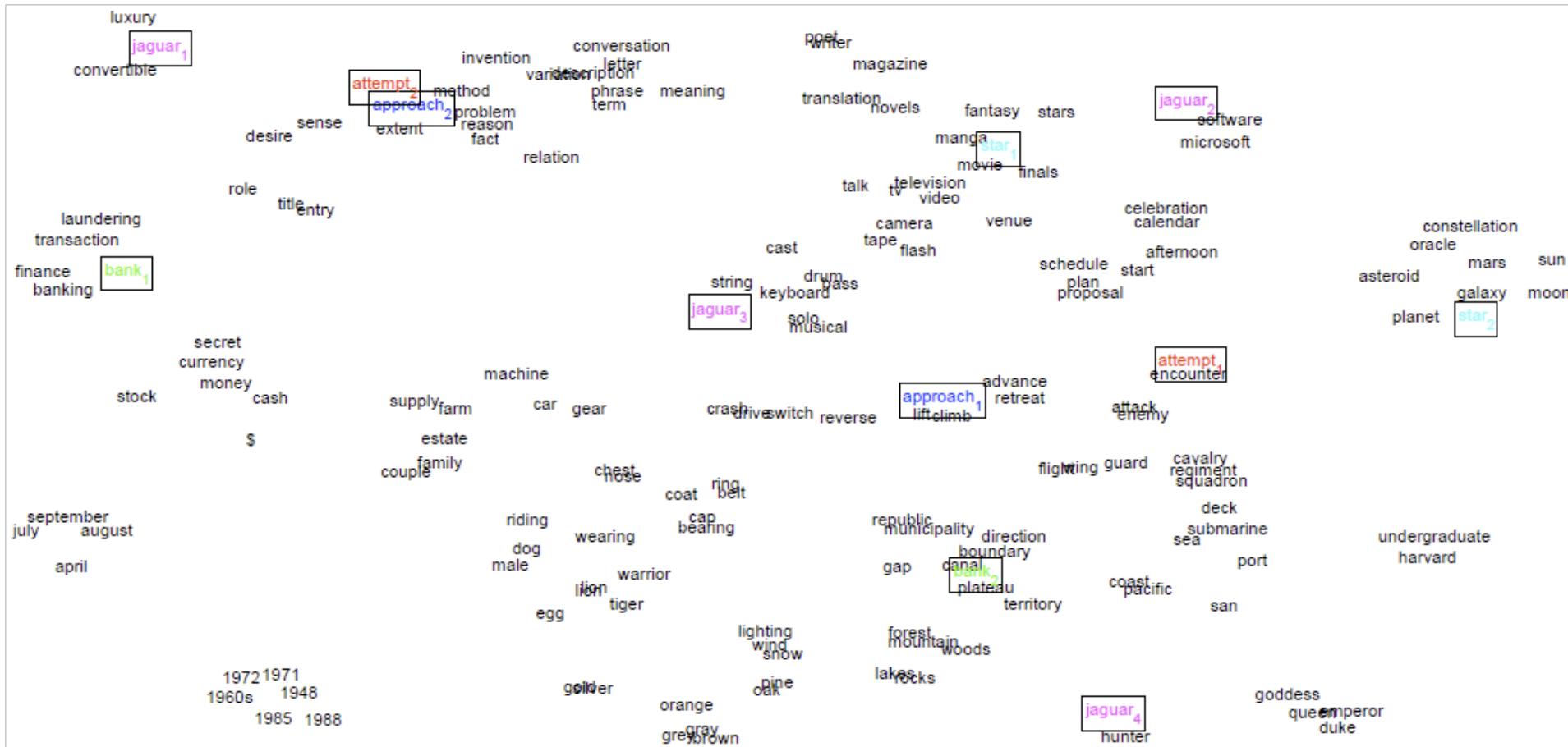
- Most words have lots of meanings!
 - Especially common words
 - Especially words that have existed for a long time
- Example: **pike**
- Does one vector capture all these meanings or do we have a mess?

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

8. Classification review and notation

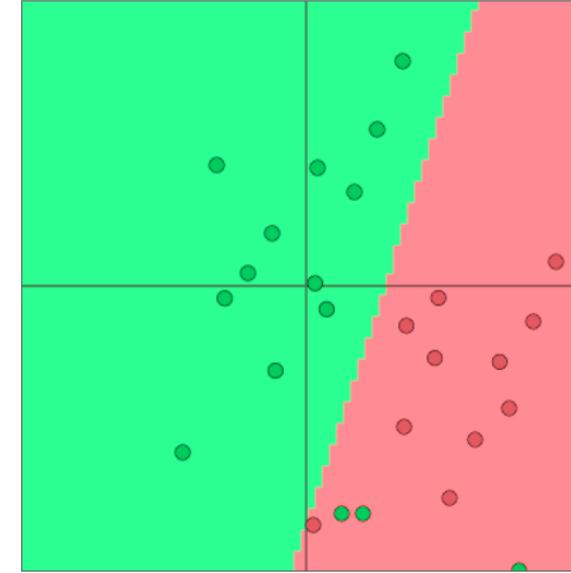
- Generally, we have a **training dataset** consisting of **samples**

$$\{x_i, y_i\}_{i=1}^N$$

- 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

Classification intuition

- Training data: $\{x_i, y_i\}_{i=1}^N$
- Simple illustration case:
 - Fixed 2D word vectors inputs to classify
 - Using softmax/logistic regression
 - Linear decision boundary
- **Traditional ML/Stats approach:** assume x_i are fixed, train (i.e., set) softmax/logistic regression weights $W \in \mathbb{R}^{C \times d}$ to determine a decision boundary (hyperplane) as in the picture
- **Method:** For each fixed x , predict:



Visualizations with ConvNetJS by Andrej Karpathy!
<http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html>

$$p(y|x) = \frac{\exp(W_y \cdot x)}{\sum_{c=1}^C \exp(W_c \cdot x)}$$

Softmax classifier

$$p(y|x) = \frac{\exp(W_y \cdot x)}{\sum_{c=1}^C \exp(W_c \cdot x)}$$

Again, we can tease apart the prediction function into three steps:

1. For each row y row of W , calculate dot product with x : $W_y \cdot x = \sum_{i=1}^d W_{yi}x_i = f_y$
2. Apply softmax function to get normalized probability:

$$p(y|x) = \frac{\exp(f_y)}{\sum_{c=1}^C \exp(f_c)} = \text{softmax}(f_y)$$

3. Choose the y with maximum probability
 - For each training example (x,y) , our objective is to **maximize the probability of the correct class y** or we can **minimize the negative log probability of that class**:

$$-\log p(y|x) = -\log \left(\frac{\exp(f_y)}{\sum_{c=1}^C \exp(f_c)} \right)$$

Training with “cross entropy loss”

- Concept of “cross entropy” is from information theory
- Let the true probability distribution be p
- Let 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:
 $p = [0, \dots, 0, 1, 0, \dots, 0]$ then:
- **Because of one-hot p , the only term left is the negative log probability of the true class: $-\log p(y_i|x_i)$**

Classification over a full dataset

- Cross entropy loss function over full dataset $\{x_i, y_i\}_{i=1}^N$

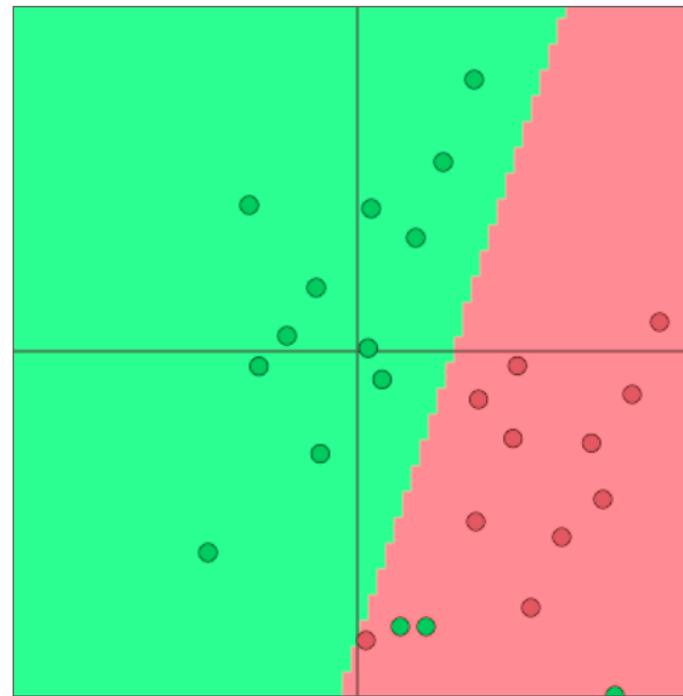
$$J(\theta) = \frac{1}{N} \sum_{i=1}^N -\log \left(\frac{e^{f_{y_i}}}{\sum_{c=1}^C e^{f_c}} \right)$$

- Instead of

$$f_y = f_y(x) = W_y \cdot x = \sum_{j=1}^d W_{yj} x_j$$

We will write f in matrix notation:

$$f = Wx$$



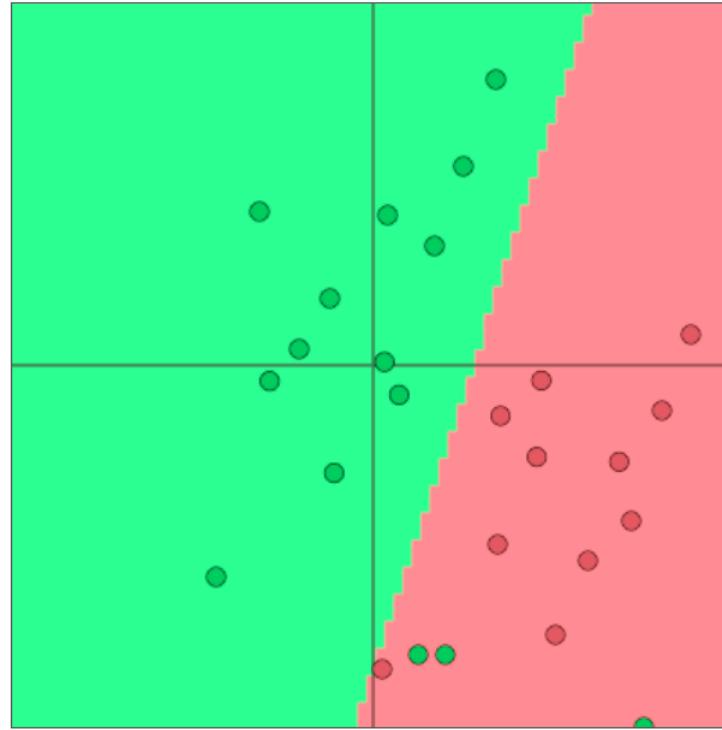
Traditional ML optimization

- For statistical machine learning θ usually only consists of the elements of W :

$$\theta = \begin{bmatrix} W_{\cdot 1} \\ \vdots \\ W_{\cdot d} \end{bmatrix} = W(:) \in \mathbb{R}^{Cd}$$

- So, we update the decision boundary via only updating W

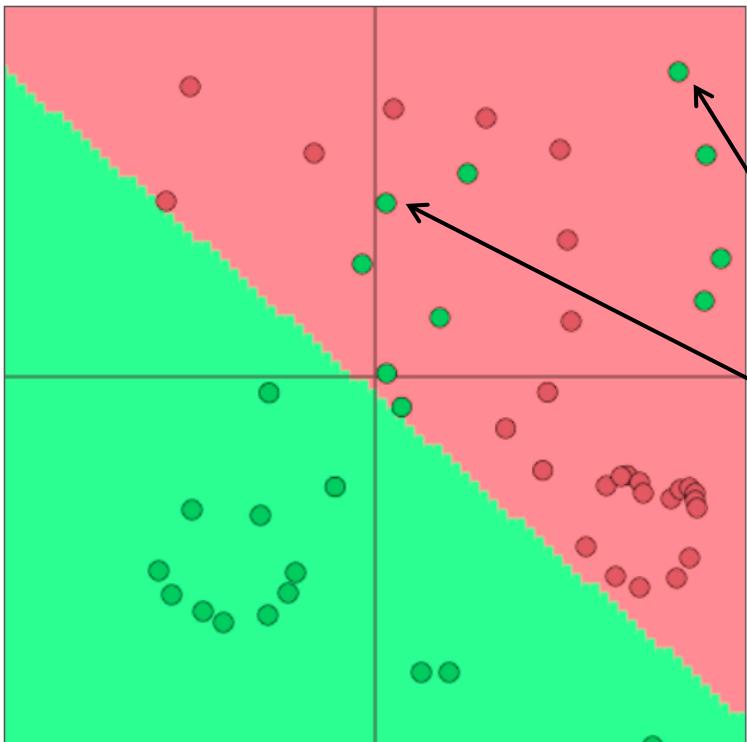
$$\nabla_{\theta} J(\theta) = \begin{bmatrix} \nabla_{W_{\cdot 1}} \\ \vdots \\ \nabla_{W_{\cdot d}} \end{bmatrix} \in \mathbb{R}^{Cd}$$



Visualizations with ConvNetJS by Karpathy

9. Neural Network Classifiers

- Softmax (\approx logistic regression) alone is not very powerful
- Softmax classifier only gives linear decision boundaries



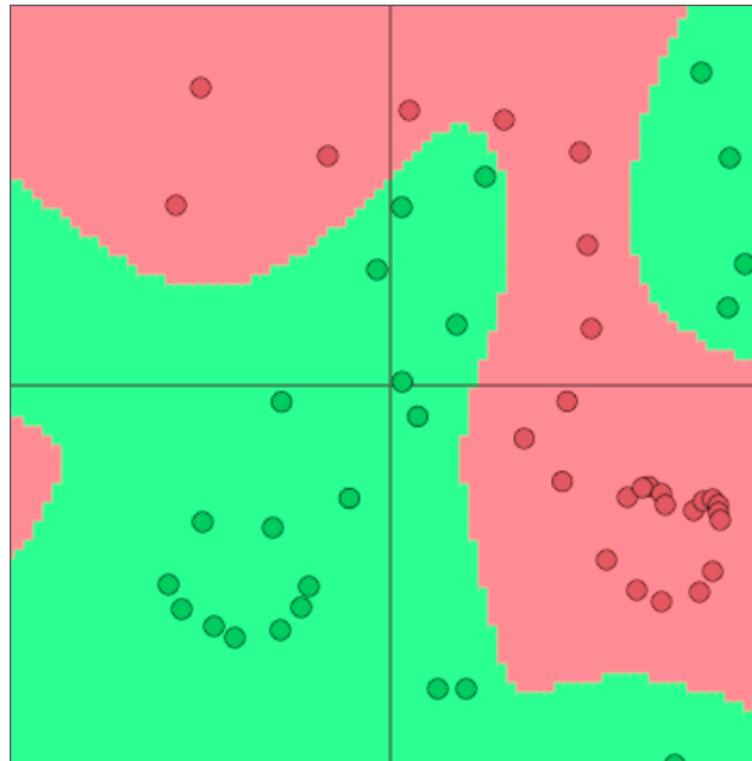
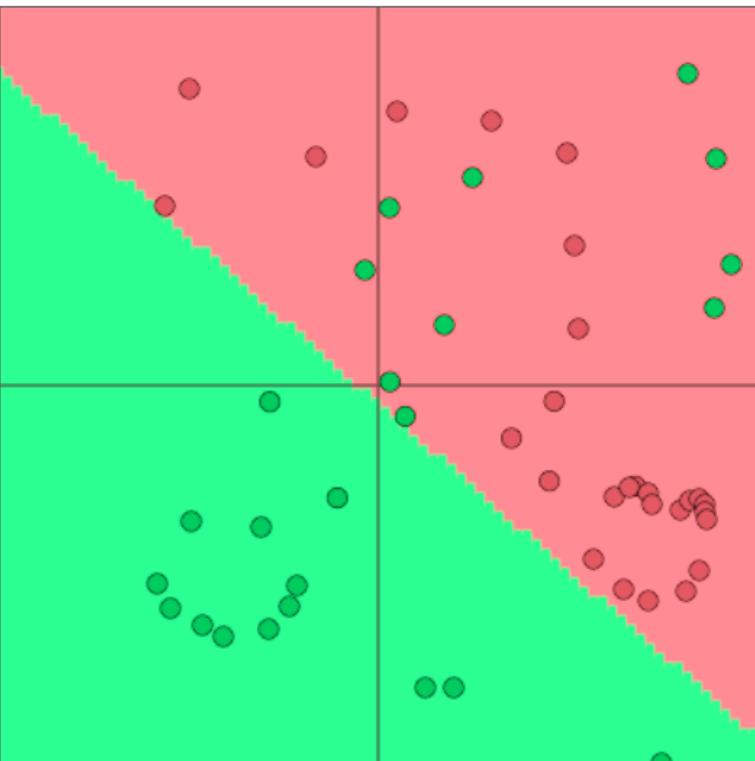
This can be quite limiting

→ Unhelpful when a problem is complex

Wouldn't it be cool to get these correct?

Neural Nets for the Win!

- Neural networks can learn much more complex functions with nonlinear decision boundaries!
 - Non-linear in the original space



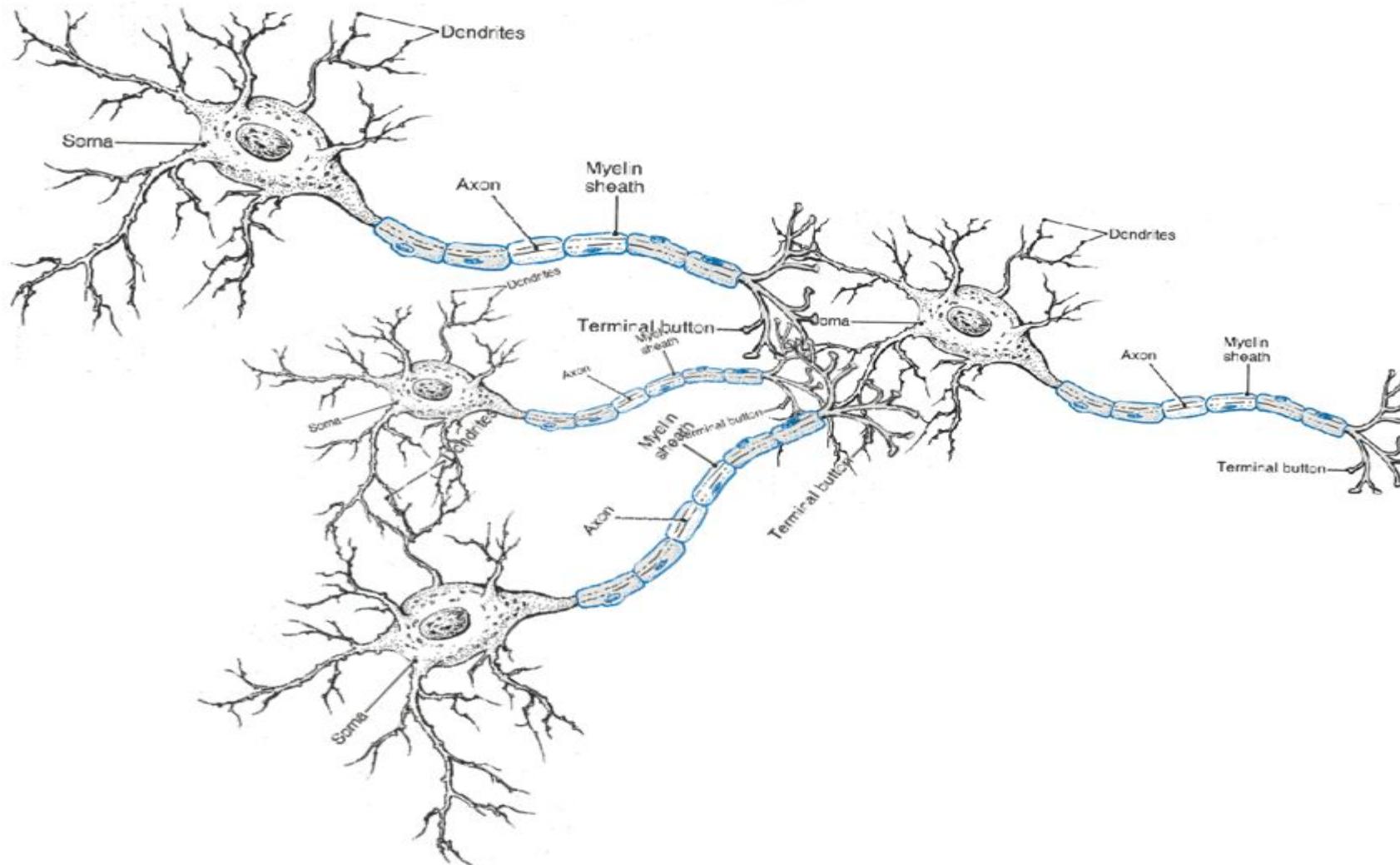
Classification difference with word vectors #1

- Commonly in NLP deep learning:
 - We learn **both** W and word vectors x
 - We learn **both** conventional parameters **and** (**distributed!**) representations
 - The word vectors re-represent one-hot vectors—they move them around in an intermediate layer vector space—for easy classification with a (linear) softmax classifier, conceptually via an embedding layer: $x = Le$

$$\nabla_{\theta} J(\theta) = \begin{bmatrix} \nabla_{W_{.1}} \\ \vdots \\ \nabla_{W_{.d}} \\ \nabla_{x_{ardvark}} \\ \vdots \\ \nabla_{x_{zebra}} \end{bmatrix} \in \mathbb{R}^{Cd+Vd}$$

Very large number of parameters!

Neural computation



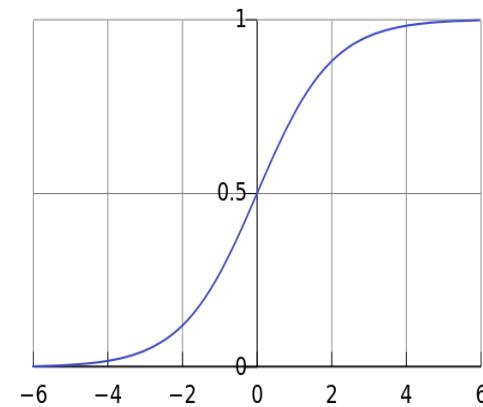
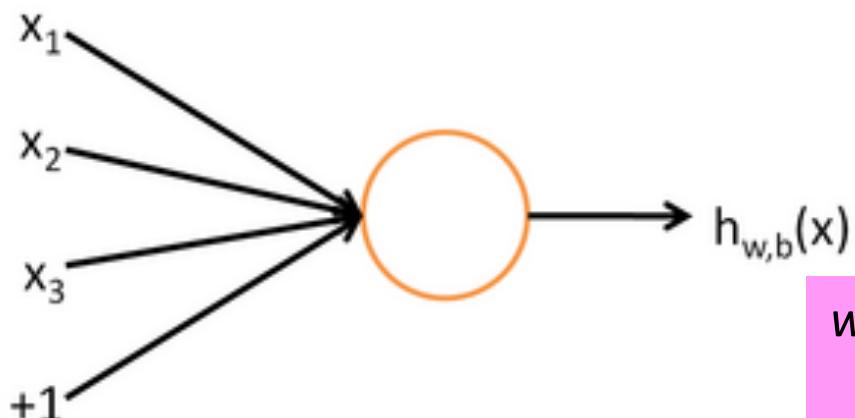
A neuron can be modeled as a binary logistic regression unit

f = nonlinear activation function (e.g. sigmoid), w = weights, b = bias, h = hidden, x = inputs

$$h_{w,b}(x) = f(w^\top x + b)$$

b : We can have an “always on” bias feature, which gives a class prior, or separate it out, as a bias term

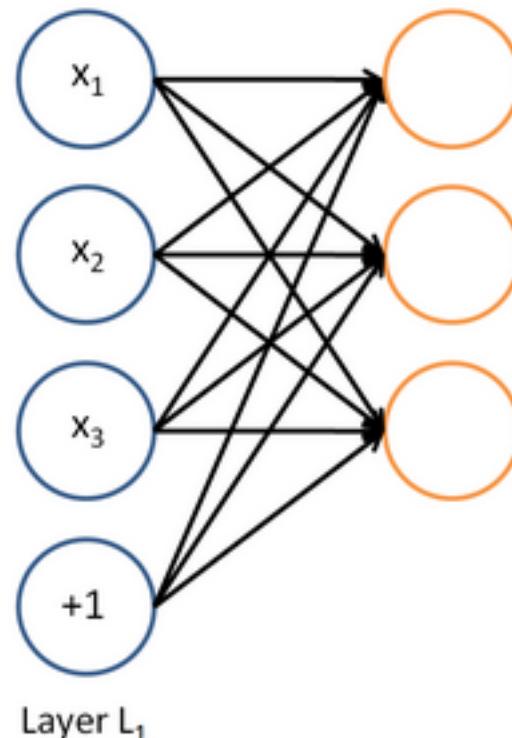
$$f(z) = \frac{1}{1 + e^{-z}}$$



w , b are the parameters of this neuron
i.e., this logistic regression model

Difference #2: A neural network = running several logistic regressions at the same time

If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs ...

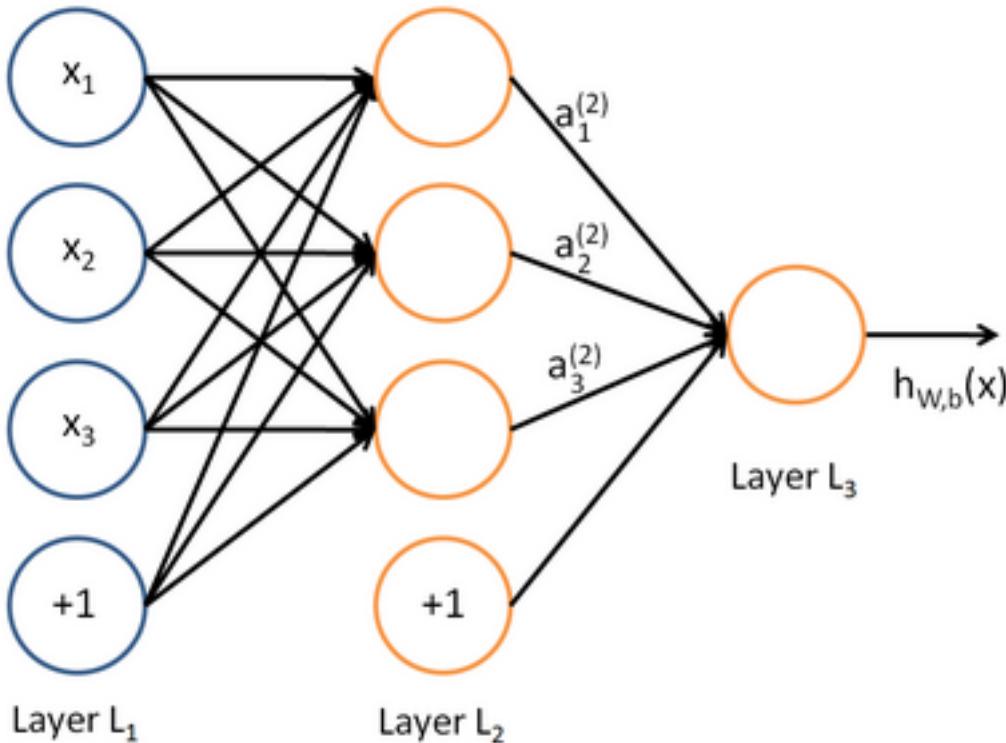


But we don't have to decide ahead of time what variables these logistic regressions are trying to predict!

Difference #2: A neural network

= running several logistic regressions at the same time

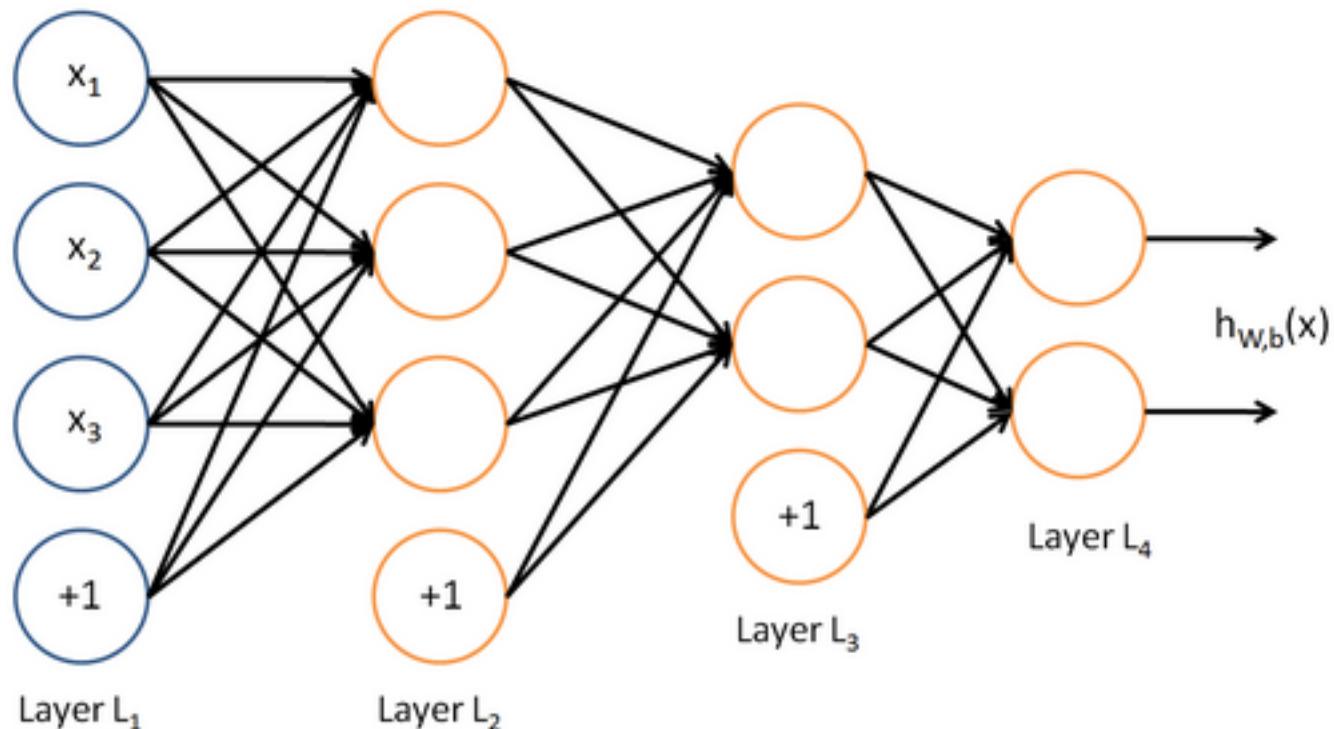
... which we can feed into another logistic regression function



It is the loss function that will direct what the intermediate hidden variables should be, so as to do a good job at predicting the targets for the next layer, etc.

Difference #2: A neural network = running several logistic regressions at the same time

Before we know it, we have a multilayer neural network....



Matrix notation for a layer

We have

$$a_1 = f(W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1)$$

$$a_2 = f(W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_2)$$

etc.

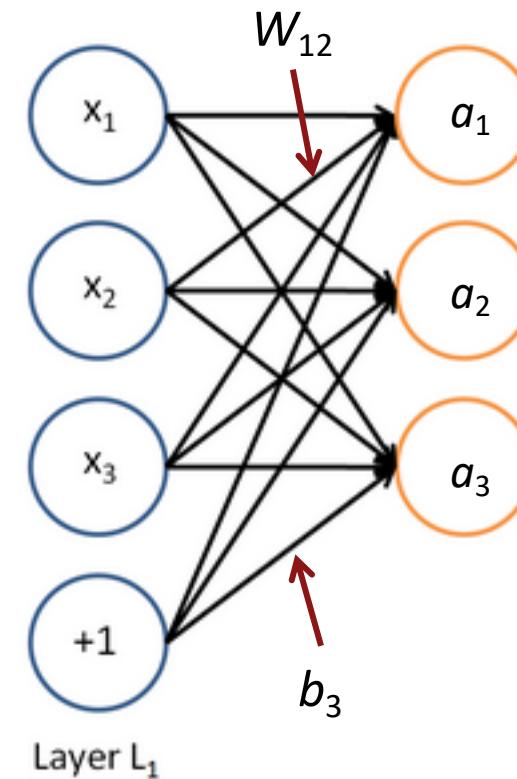
In matrix notation

$$z = Wx + b$$

$$a = f(z)$$

Activation f is applied element-wise:

$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$$



Non-linearities (aka “ f ”): Why they’re needed

- Example: function approximation, e.g., regression or classification
 - Without non-linearities, deep neural networks can’t do anything more than a linear transform
 - Extra layers could just be compiled down into a single linear transform: $W_1 W_2 x = Wx$
 - With more layers, they can approximate more complex functions!

