Natural Language Processing with Deep Learning

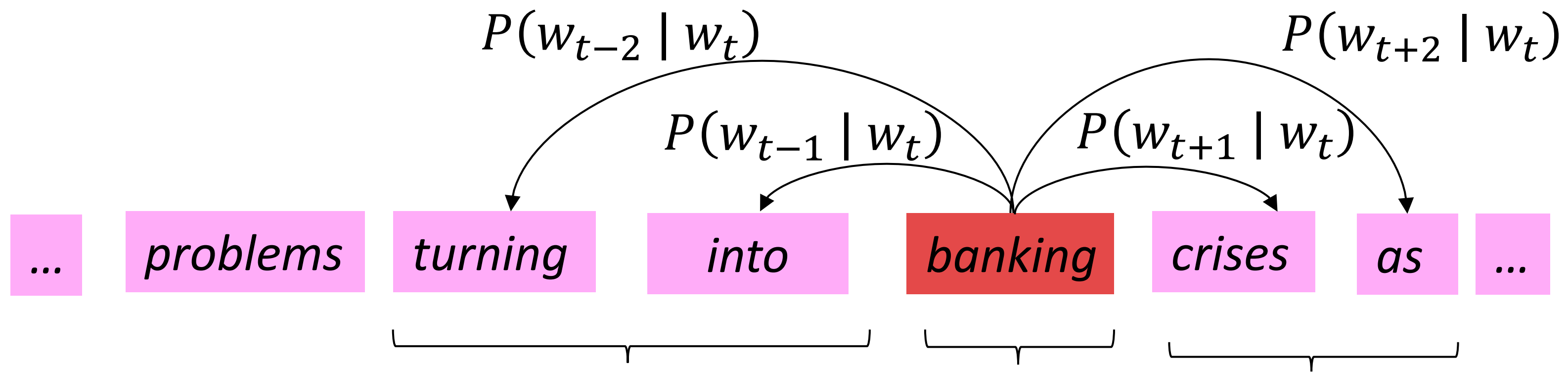
CS224N/Ling284

Christopher Manning

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

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!

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

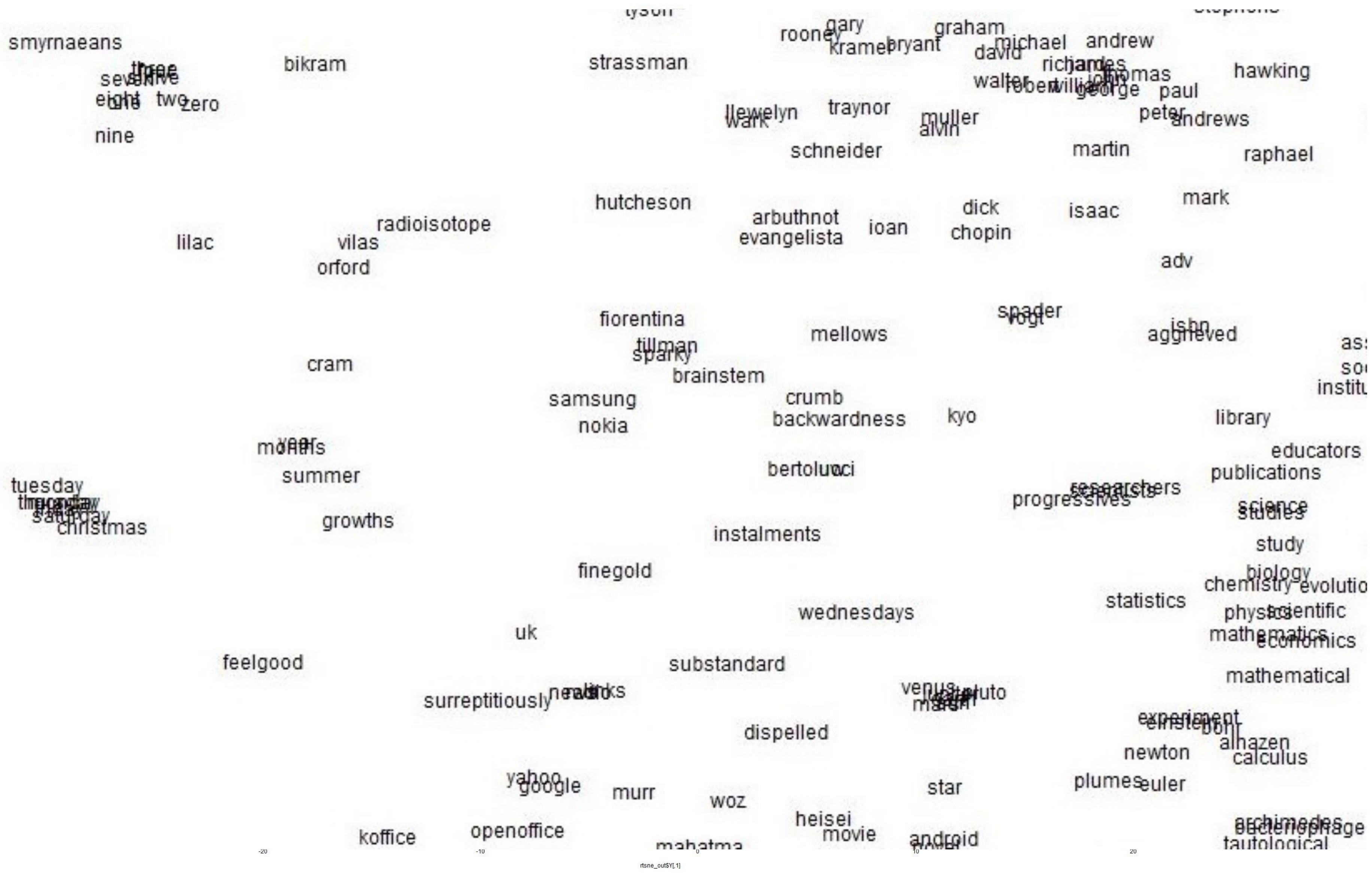
$U.{v_4}^T$ dot product

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

“Bag of words” model! $\longrightarrow$ 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



The skip-gram model with negative sampling (HW2)

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

- $$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$ $\leftarrow$ A big sum over words

- Hence, in standard word2vec and HW2 you implement 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)

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

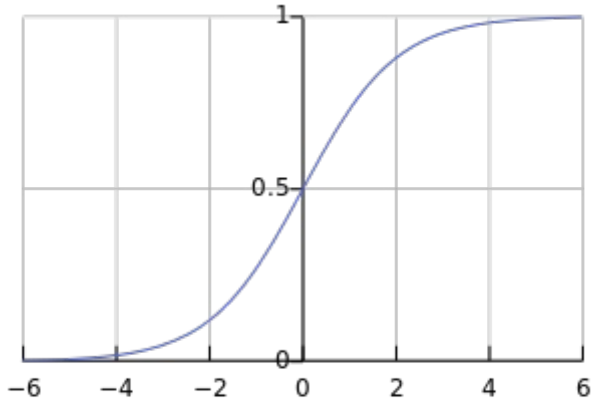
So far, we explained **naïve softmax**

The skip-gram model with negative sampling (HW2)

- Introduced in: "Distributed Representations of Words and Phrases and their Compositionality" (Mikolov et al. 2013)
- Overall objective function (they maximize): $J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J_t(\theta)$

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

- 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 in second part



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!

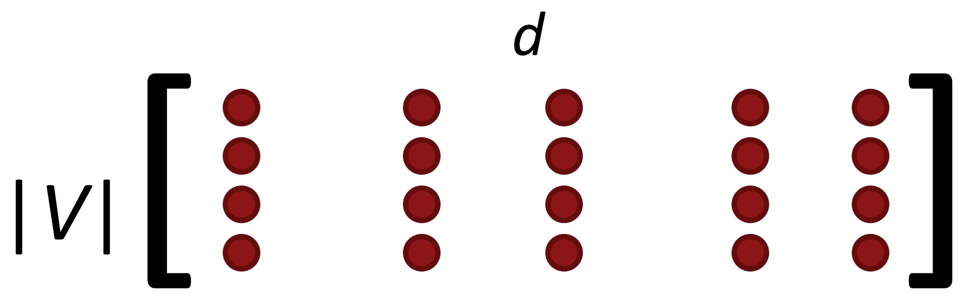
$$\nabla_\theta J_t(\theta) = \begin{bmatrix} 0 \\ \vdots \\ \nabla_{v_{like}} \\ \vdots \\ 0 \\ \nabla_{u_I} \\ \vdots \\ \nabla_{u_{learning}} \\ \vdots \end{bmatrix} \in \mathbb{R}^{2dV}$$

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

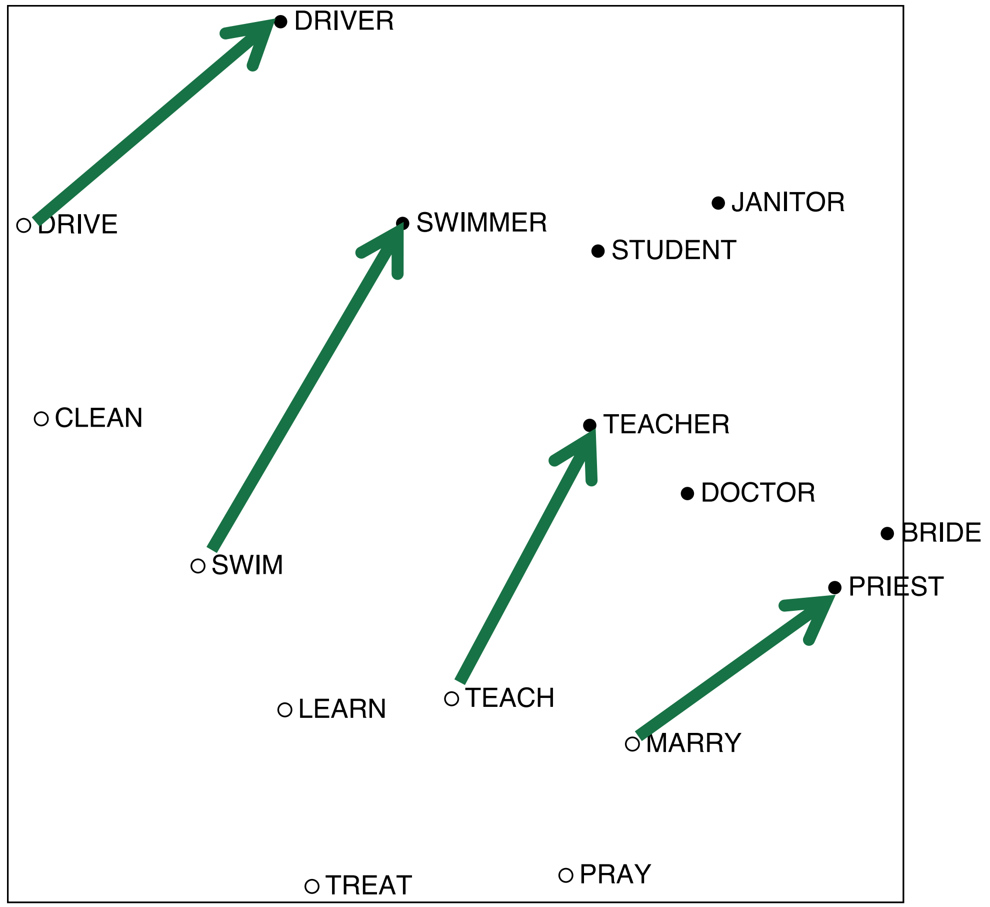
Rows not columns in actual DL packages!



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

This is also a particular issue with more advanced optimization methods in the Adagrad family

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

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 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!

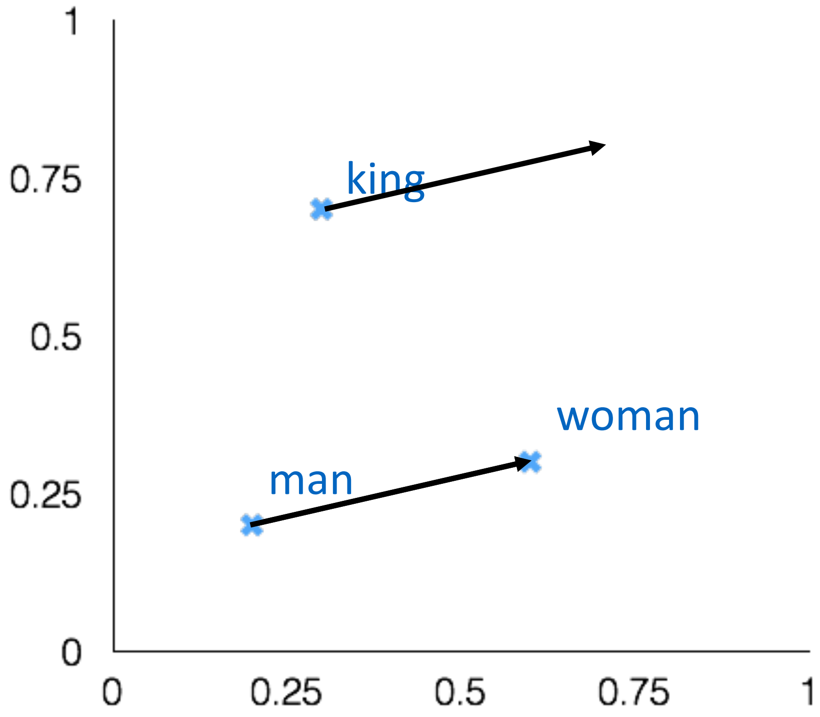
Intrinsic word vector evaluation

- Word Vector Analogies

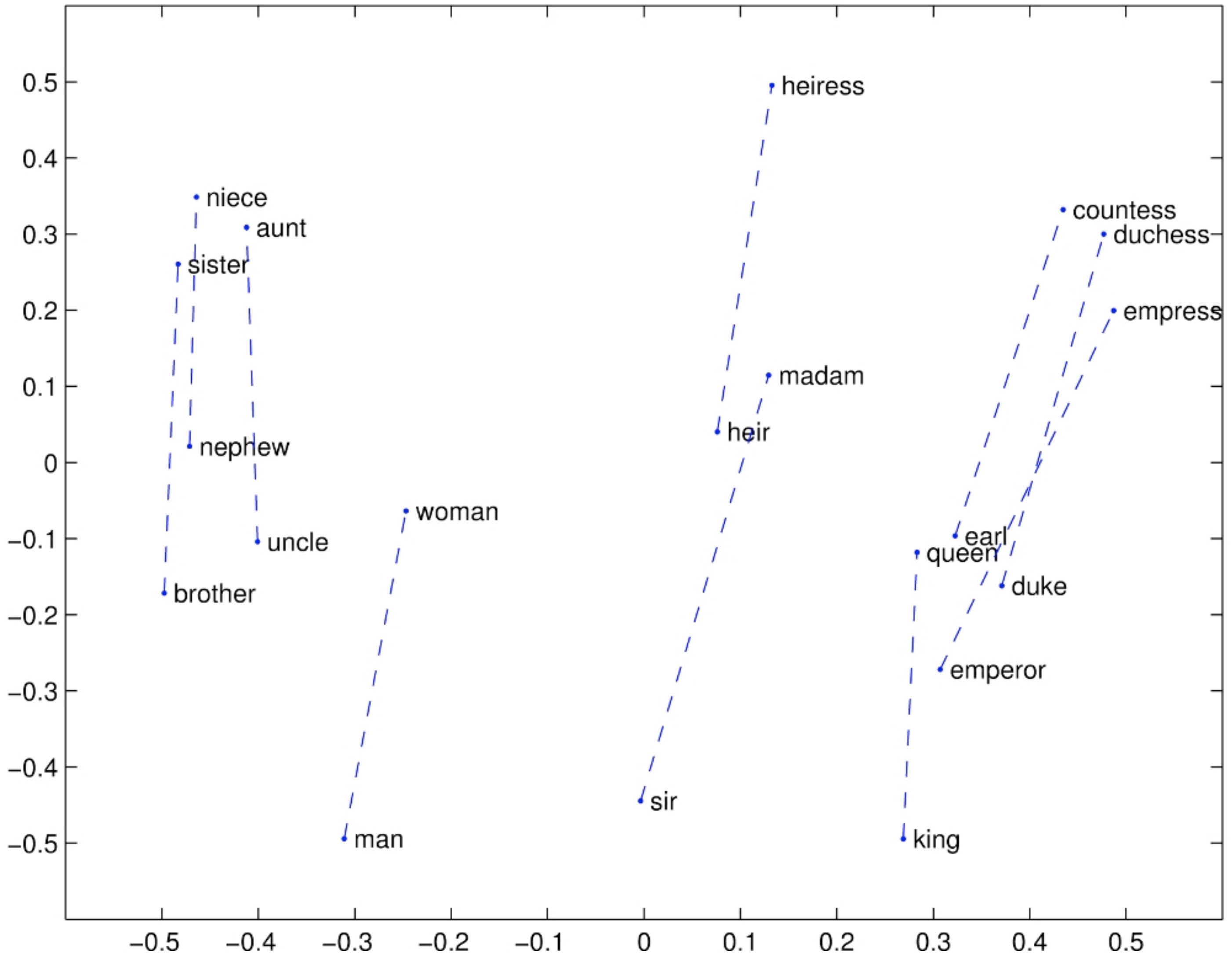
$$\boxed{a:b :: c:?} \longrightarrow \boxed{d = \arg\max_i \frac{(x_b - x_a + x_c)^T x_i}{||x_b - x_a + x_c||}}$$

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

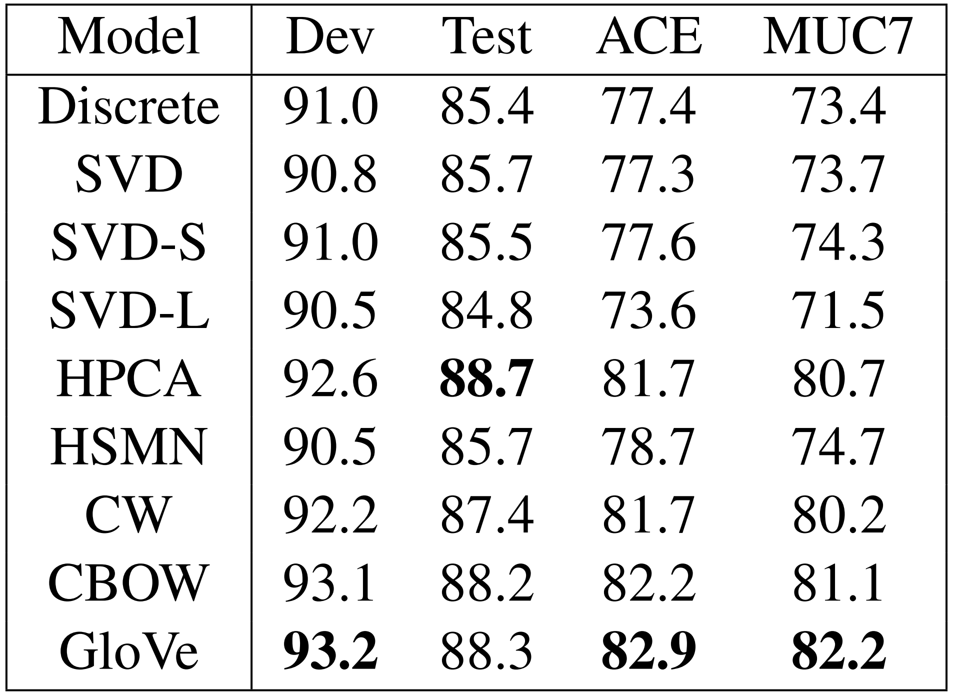
- Word vector distances and their correlation with human judgments

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	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

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

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**

- Subsequent NLP tasks in this class are other examples. So, more examples soon.

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**

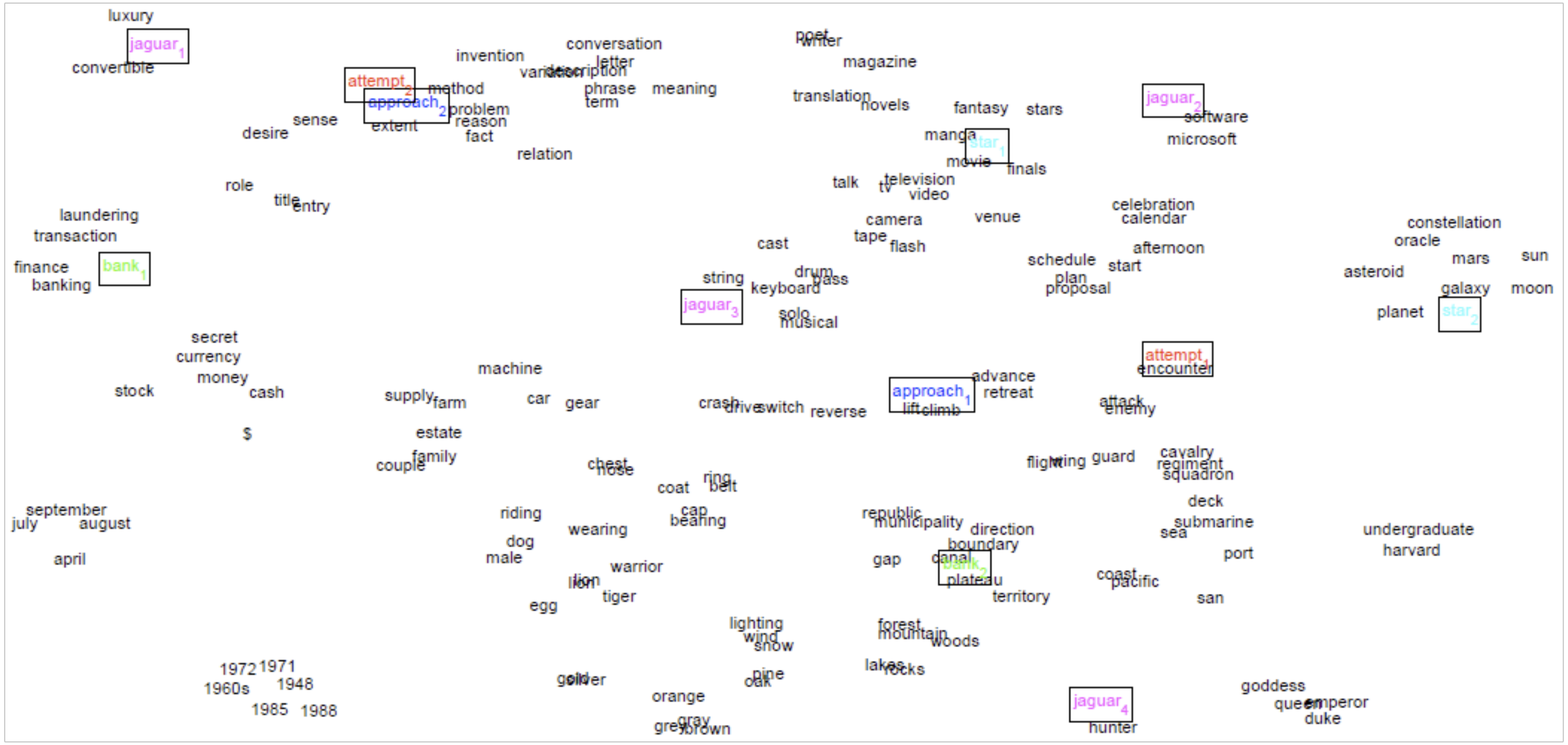
- Does one vector capture all these meanings or do we have a mess?

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!*

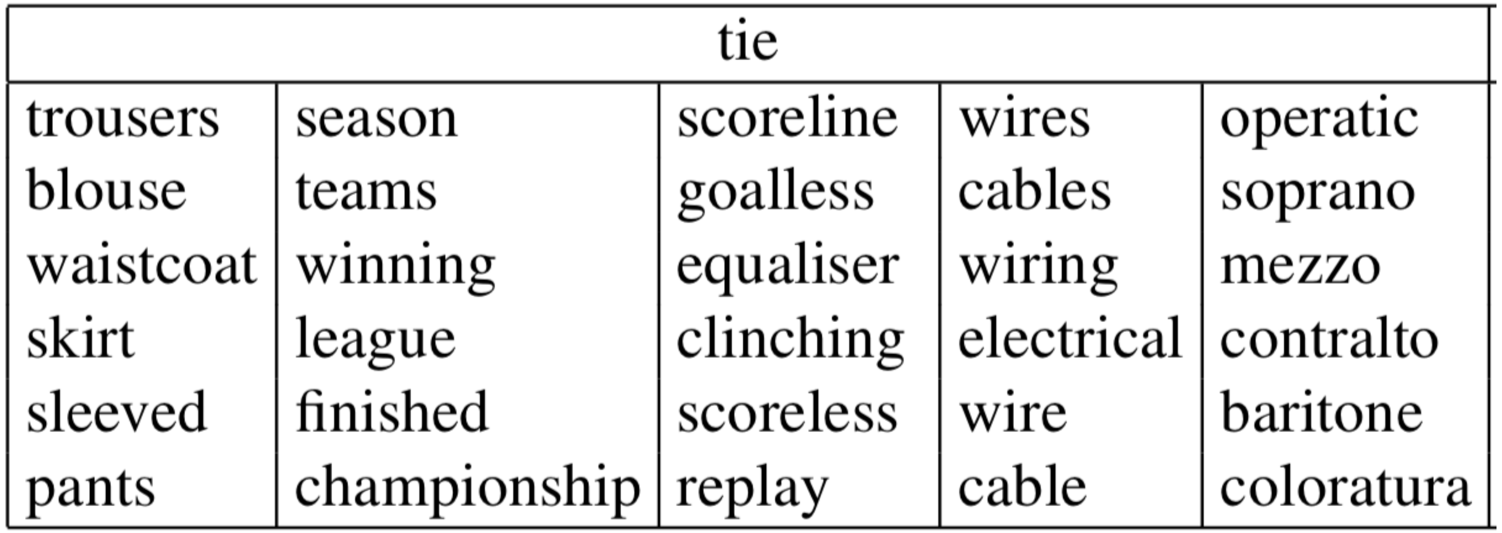
Improving Word Representations Via Global Context And Multiple Word Prototypes (Huang et al. 2012)

- 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

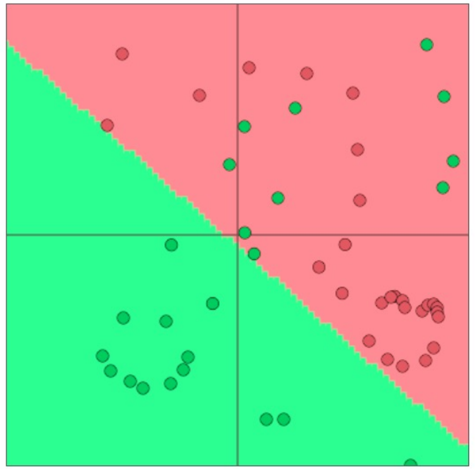
$$\{x_i, y_i\}^N_{i=1}$$

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

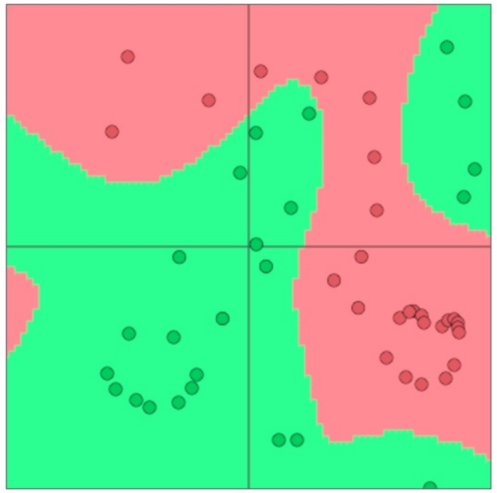
- y_i are labels (one of C classes) we try to predict, for example:
  - classes: sentiment (+/–), named entities, buy/sell decision
  - other words
  - later: multi-word sequences

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

Softmax classifier

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

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

1. For each row *y* of *W,* calculate dot product with *x*: $$W_y.x = \sum_{i=1}^{d} W_{yi}x_i = f_y$$

2. Apply softmax function to get normalized probability:

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

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:

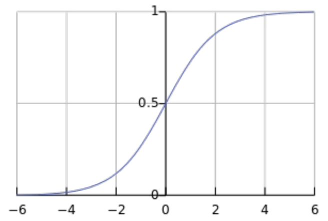
$$-\log p(y|x) = -\log\left(\frac{\exp(f_y)}{\sum_{c=1}^{C} \exp(f_c)}\right)$$

NER: Binary classification for center word being location

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

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

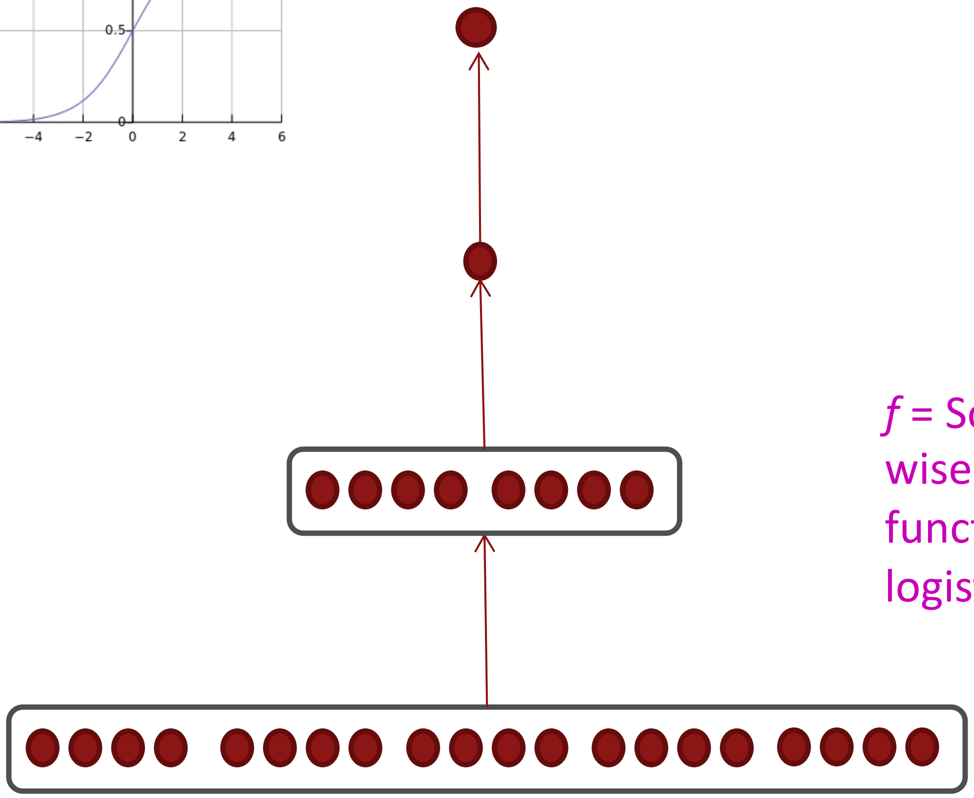
predicted model probability of class



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

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

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



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

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:

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

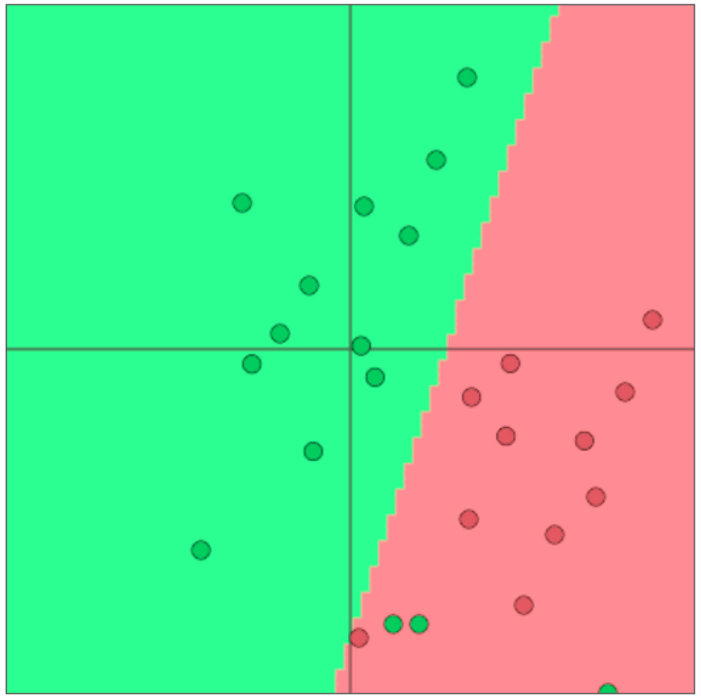
- Assuming a ground truth (or true or gold or target) probability distribution that is 1 at the right class and 0 everywhere else, *p* = [0, …, 0, 1, 0, …, 0], then:
- **Because of one-hot *p*, the only term left is the negative log probability of the true class y_i: $-\log p(y_i|x_i)$**

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

Classification over a full dataset

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

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



Remember: Stochastic Gradient Descent

Update equation:

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

α = *step size or learning rate*

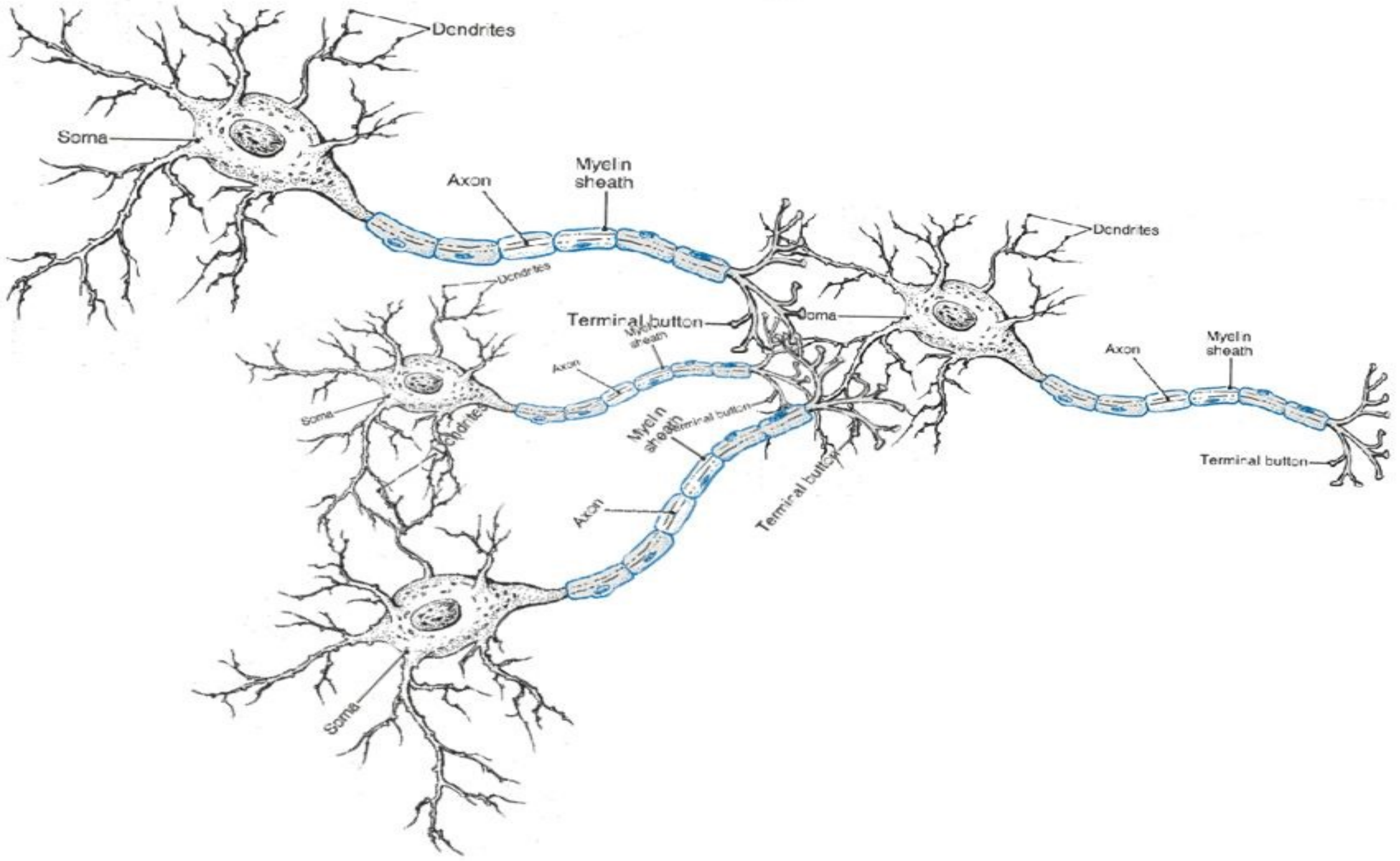
i.e., for each parameter: $\theta_j^{new} = \theta_j^{old} - \alpha \frac{\partial J(\theta)}{\partial \theta_j^{old}}$

In deep learning, θ includes the data representation (e.g., word vectors) too!

How can we compute $\nabla_{\theta} J(\theta)$?

1. By hand
2. Algorithmically: the backpropagation algorithm (next lecture!)

7. Neural computation



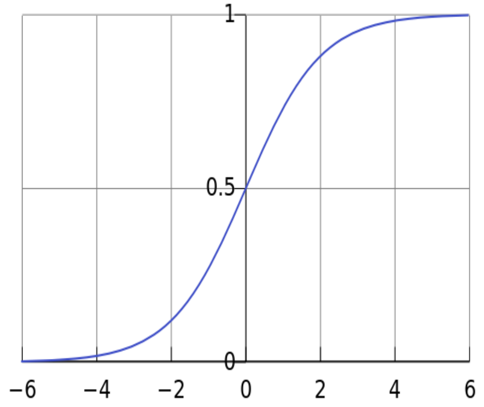
A binary logistic regression unit is a bit similar to a neuron

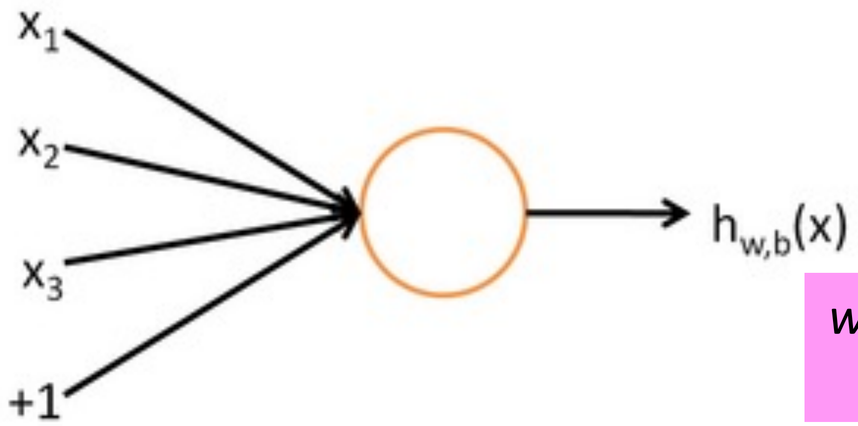
f = nonlinear activation function (e.g. sigmoid), *w* = weights, *b* = bias, *h* = hidden, *x* = inputs

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

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

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

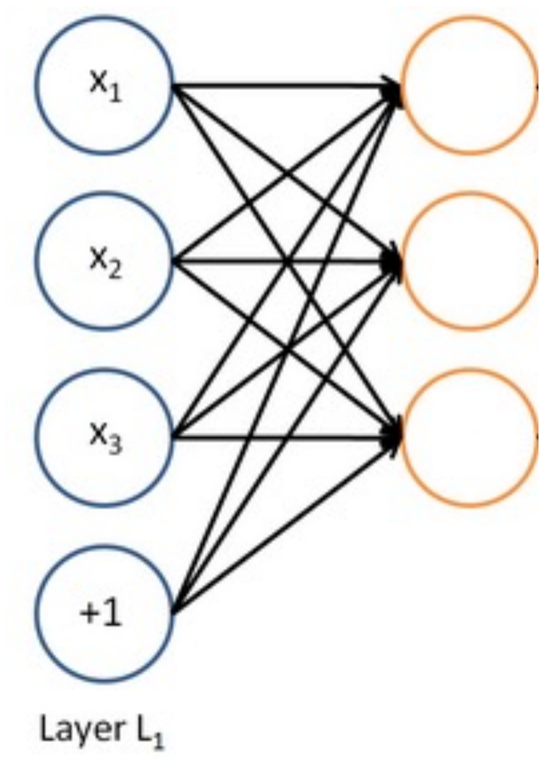




w, *b* are the parameters of this neuron i.e., this logistic regression model

A neural network
= running several logistic regressions at the same time

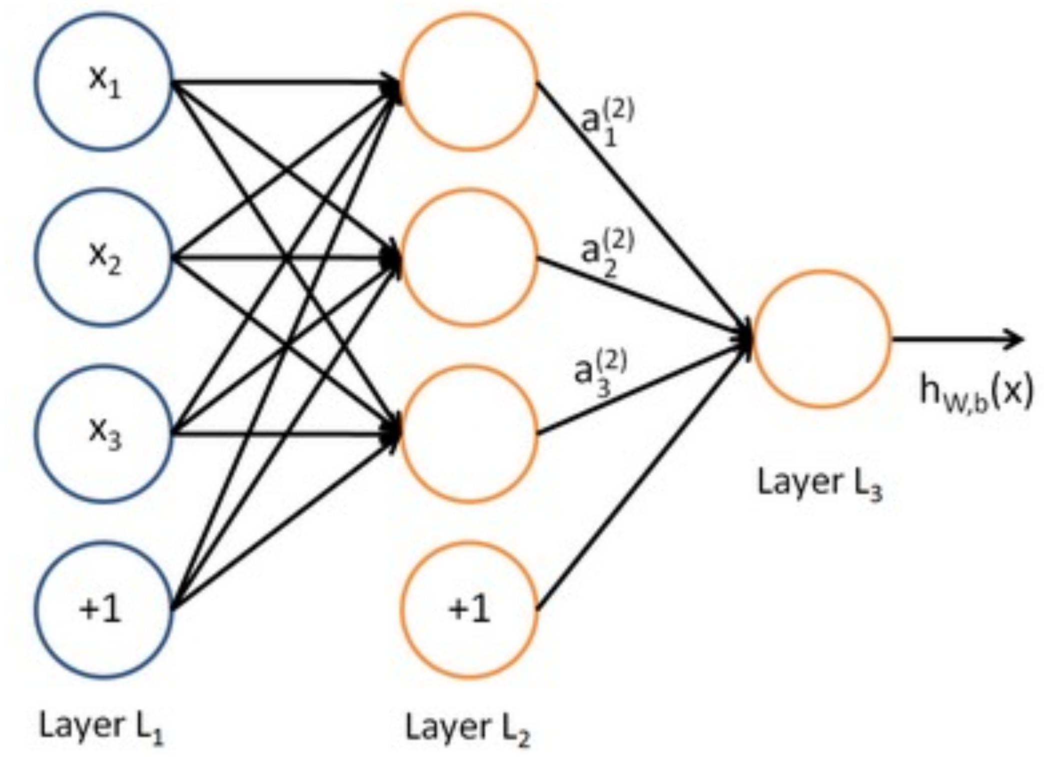
If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs …



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

A neural network
= running several logistic regressions at the same time

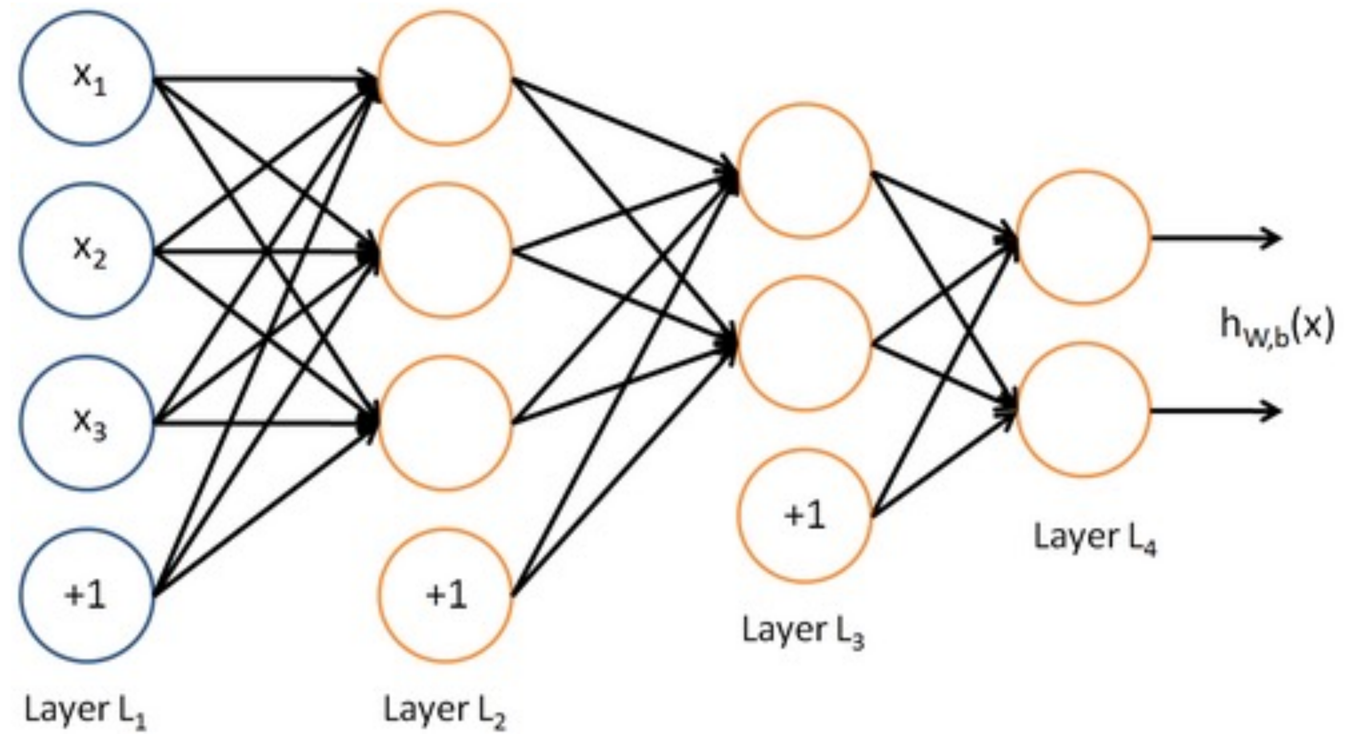
… which we can feed into another logistic regression function, giving composed functions



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.

A neural network
= running several logistic regressions at the same time

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



This allows us to re-represent and compose our data multiple times and to learn a classifier that is highly non-linear in terms of the original inputs
(but typically is linear in terms of the pre-final layer representations)

Matrix notation for a layer

We have

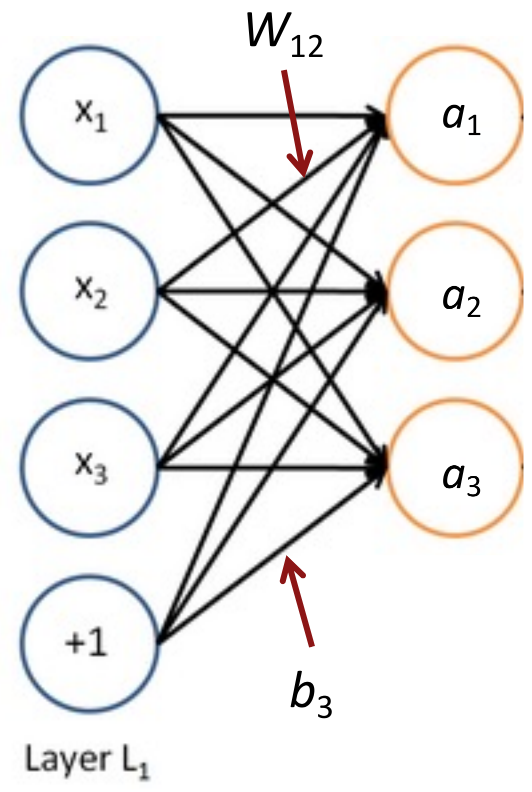
$$a_1 = f(W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1)$$
$$a_2 = f(W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_2)$$
etc.

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 (like *f* or sigmoid): Why they're needed

- Neural networks do 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: $W_1 W_2 x = Wx$
  - But, with more layers that include non-linearities, they can approximate more complex functions!