

How do we represent the meaning of a word?

Definition: **Meaning** (Webster dictionary)

- the idea that is represented by a word, phrase, etc.
- the idea that a person wants to express by using words, signs, etc.
- the idea that is expressed in a work of writing, art, etc.

How to represent meaning in a computer?

Common answer: Use a taxonomy like WordNet that has hypernyms (is-a) relationships and

```
from nltk.corpus import wordnet as wn
panda = wn.synset('panda.n.01')
hyper = lambda s: s.hypernyms()
list(panda.closure(hyper))
```

synonym sets (good):

```
[Synset('procyonid.n.01'),
Synset('carnivore.n.01'),
Synset('placental.n.01'),
Synset('mammal.n.01'),
Synset('vertebrate.n.01'),
Synset('chordate.n.01'),
Synset('animal.n.01'),
Synset('organism.n.01'),
Synset('living_thing.n.01'),
Synset('whole.n.02'),
Synset('object.n.01'),
Synset('physical_entity.n.01'),
Synset('entity.n.01')]
```

```
S: (adj) full, good
S: (adj) estimable, good, honorable, respectable
S: (adj) beneficial, good
S: (adj) good, just, upright
S: (adj) adept, expert, good, practiced,
proficient, skillful
S: (adj) dear, good, near
S: (adj) good, right, ripe
...
S: (adv) well, good
S: (adv) thoroughly, soundly, good
S: (n) good, goodness
S: (n) commodity, trade good, good
```

Problems with this discrete representation

- Great as resource but missing nuances, e.g.
synonyms:
adept, expert, good, practiced, proficient, skillful?
- Missing new words (impossible to keep up to date):
wicked, badass, nifty, crack, ace, wizard, genius, ninja
- Subjective
- Requires human labor to create and adapt
- Hard to compute accurate word similarity →

Problems with this discrete representation

The vast majority of rule-based **and** statistical NLP work regards words as atomic symbols: *hotel, conference, walk*

In vector space terms, this is a vector with one 1 and a lot of zeroes

$[0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0]$

Dimensionality: 20K (speech) – 50K (PTB) – 500K (big vocab) – 13M (Google 1T)

We call this a “one-hot” representation. Its problem:

motel $[0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0]$ AND
hotel $[0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0]$ = 0

Distributional similarity based representations

You can get a lot of value by representing a word by means of its neighbors

“You shall know a word by the company it keeps”

(J. R. Firth 1957: 11)

One of the most successful ideas of modern statistical NLP

government debt problems turning into banking crises as has happened in
saying that Europe needs unified banking regulation to replace the hodgepodge

↖ These words will represent *banking* ↗

How to make neighbors represent words?

Answer: With a cooccurrence matrix X

- 2 options: full document vs windows
- Word - document cooccurrence matrix will give general topics (all sports terms will have similar entries) leading to “Latent Semantic Analysis”
- Window allows us to capture both syntactic (POS) and semantic information →

Window based cooccurrence 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.

Window based cooccurrence matrix

- 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

Problems with simple cooccurrence vectors

Increase in size with vocabulary

Very high dimensional: require a lot of storage

Subsequent classification models have sparsity issues

→ Models are less robust

Solution: Low dimensional vectors

- Idea: store “most” of the important information in a fixed, small number of dimensions: a dense vector
- Usually around 25 – 1000 dimensions
- How to reduce the dimensionality?

Method 1: Dimensionality Reduction on X

Singular Value Decomposition of cooccurrence matrix X .

$$\begin{array}{cccc}
 \begin{array}{c} m \\ \boxed{} \\ n \\ X \end{array} & = & \begin{array}{c} r \\ \boxed{\begin{array}{c} | \\ | \\ | \\ U_1 U_2 U_3 \cdots \\ | \\ | \\ | \end{array}} \\ n \\ U \end{array} & \begin{array}{c} r \\ \boxed{\begin{array}{c} S_1 \quad \quad \quad 0 \\ S_2 \quad S_3 \quad \cdots \\ 0 \quad \quad \quad \ddots \\ \quad \quad \quad \quad \quad S_r \end{array}} \\ r \\ S \end{array} & \begin{array}{c} m \\ \boxed{\begin{array}{c} \hline V_1 \hline \hline V_2 \hline \hline V_3 \hline \hline \vdots \hline \hline \end{array}} \\ r \\ V^T \end{array} \\
 \\
 \begin{array}{c} m \\ \boxed{\phantom{\hat{X}}} \\ n \\ \hat{X} \end{array} & = & \begin{array}{c} k \\ \boxed{\begin{array}{c} | \\ | \\ | \\ U_1 U_2 U_3 \cdots \\ | \\ | \\ | \end{array}} \\ n \\ \hat{U} \end{array} & \begin{array}{c} k \\ \boxed{\begin{array}{c} S_1 \quad \quad \quad 0 \\ S_2 \quad S_3 \quad \cdots \\ 0 \quad \quad \quad \ddots \\ \quad \quad \quad \quad \quad S_k \end{array}} \\ k \\ \hat{S} \end{array} & \begin{array}{c} m \\ \boxed{\begin{array}{c} \hline V_1 \hline \hline V_2 \hline \hline V_3 \hline \hline \vdots \hline \hline \end{array}} \\ k \\ \hat{V}^T \end{array}
 \end{array}$$

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

Simple SVD word vectors in Python

Corpus:

I like deep learning. I like NLP. I enjoy flying.

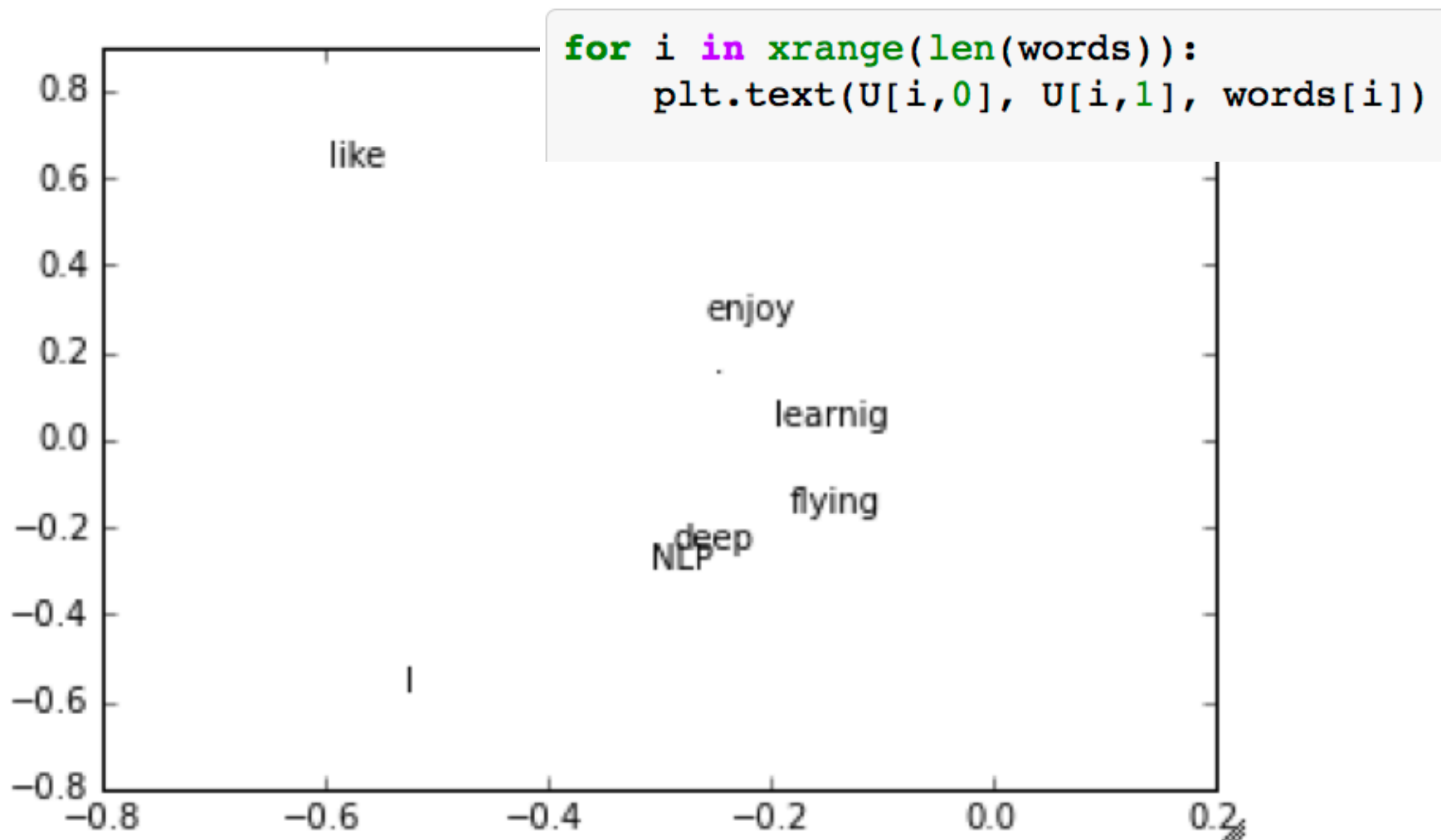
```
import numpy as np
la = np.linalg
words = ["I", "like", "enjoy",
         "deep", "learnig", "NLP", "flying", "."]
X = np.array([[0,2,1,0,0,0,0,0],
              [2,0,0,1,0,1,0,0],
              [1,0,0,0,0,0,1,0],
              [0,1,0,0,1,0,0,0],
              [0,0,0,1,0,0,0,1],
              [0,1,0,0,0,0,0,1],
              [0,0,1,0,0,0,0,1],
              [0,0,0,0,1,1,1,0]])

U, s, Vh = la.svd(X, full_matrices=False)
```

Simple SVD word vectors in Python

Corpus: I like deep learning. I like NLP. I enjoy flying.

Printing first two columns of U corresponding to the 2 biggest singular values



Word meaning is defined in terms of vectors

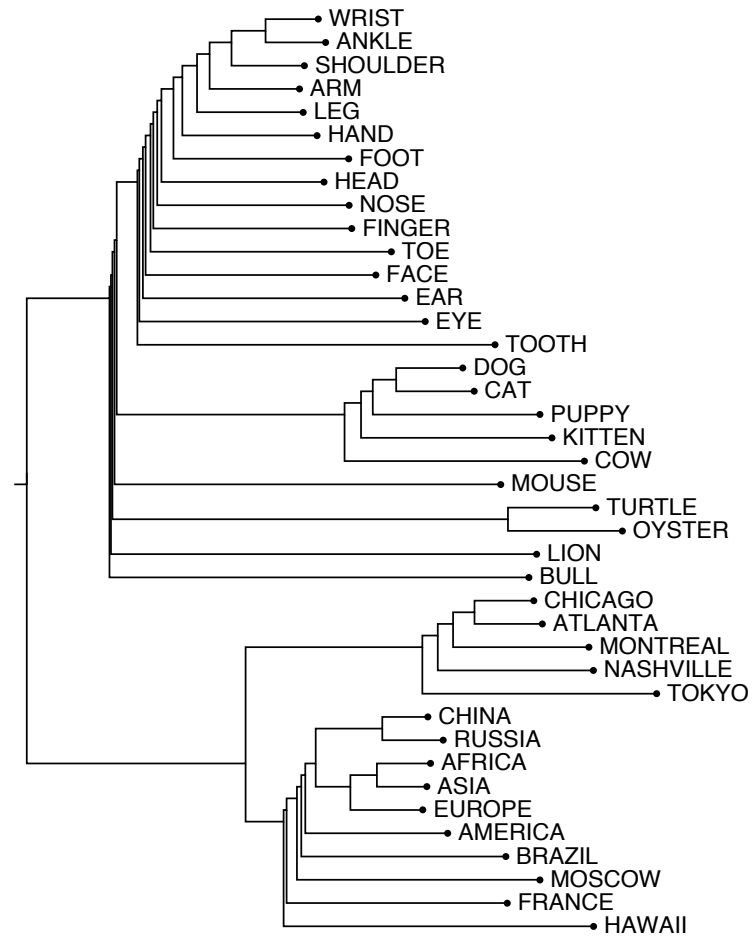
- In all subsequent models, including deep learning models, a word is represented as a dense vector

$$\textit{linguistics} = \begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \end{pmatrix}$$

Hacks to X

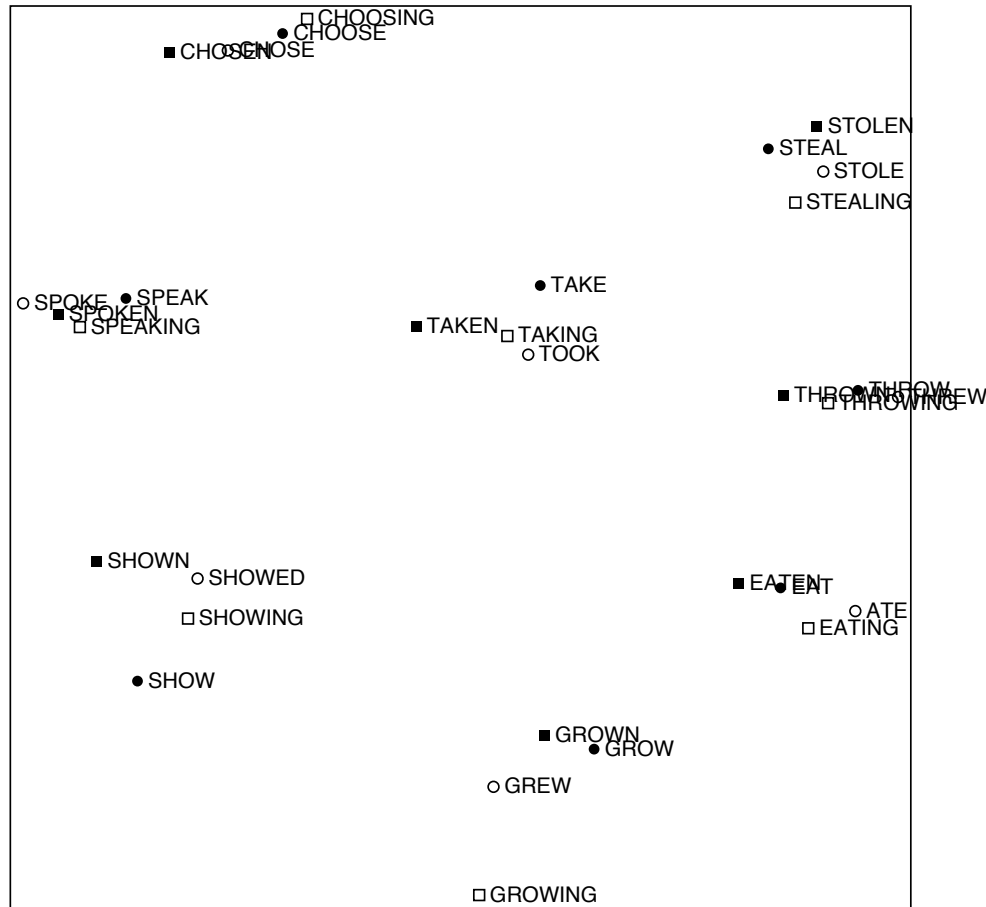
- Problem: function words (the, he, has) are too frequent → syntax has too much impact. Some fixes:
 - $\min(X,t)$, with $t \sim 100$
 - Ignore them all
- Ramped windows that count closer words more
- Use Pearson correlations instead of counts, then set negative values to 0
- +++

Interesting semantic patterns emerge in the vectors



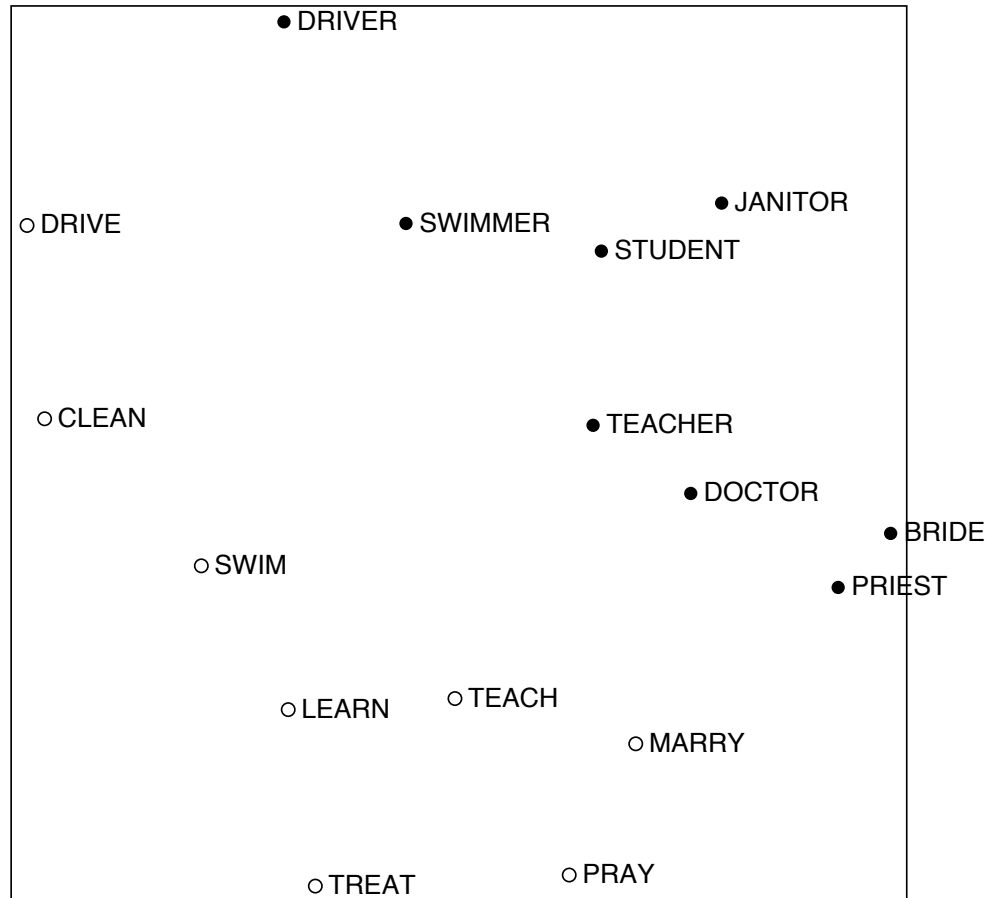
An Improved Model of Semantic Similarity Based on Lexical Co-Occurrence
Rohde et al. 2005

Interesting semantic patterns emerge in the vectors



An Improved Model of Semantic Similarity Based on Lexical Co-Occurrence
Rohde et al. 2005

Interesting semantic patterns emerge in the vectors



An Improved Model of Semantic Similarity Based on Lexical Co-Occurrence
Rohde et al. 2005

Problems with SVD

Computational cost scales quadratically for $n \times m$ matrix:

$O(mn^2)$ flops (when $n < m$)

→ Bad for millions of words or documents

Hard to incorporate new words or documents

Different learning regime than other DL models

Idea: Directly learn low-dimensional word vectors

- Old idea. Relevant for this lecture & deep learning:
 - Learning representations by back-propagating errors. (Rumelhart et al., 1986)
 - A neural probabilistic language model (Bengio et al., 2003)
 - NLP from Scratch (Collobert & Weston, 2008)
 - A recent and even simpler model: word2vec (Mikolov et al. 2013) → intro now

Main Idea of word2vec

- Instead of capturing cooccurrence counts directly,
- Predict surrounding words of every word
- Both are quite similar, see “*Glove: Global Vectors for Word Representation*” by Pennington et al. (2014)
- Faster and can easily incorporate a new sentence/document or add a word to the vocabulary

Details

- Predict surrounding words in a window of length c of every word.
- Objective function: Maximize the log probability of any context word given the current center word:

- $$J(\theta) = \frac{1}{T} \sum_{t=1}^T \sum_{-c \leq j \leq c, j \neq 0} \log p(w_{t+j} | w_t)$$

Details

- Predict surrounding words in a window of length c of every word

- For $p(w_{t+j}|w_t)$ the simplest first formulation is

$$p(w_O|w_I) = \frac{\exp\left(v'_{w_O} \top v_{w_I}\right)}{\sum_{w=1}^W \exp\left(v'_w \top v_{w_I}\right)}$$

- where v and v' are “input” and “output” vector representations of w (so every word has two vectors!)
- This is essentially “dynamic” logistic regression

Cost/Objective functions

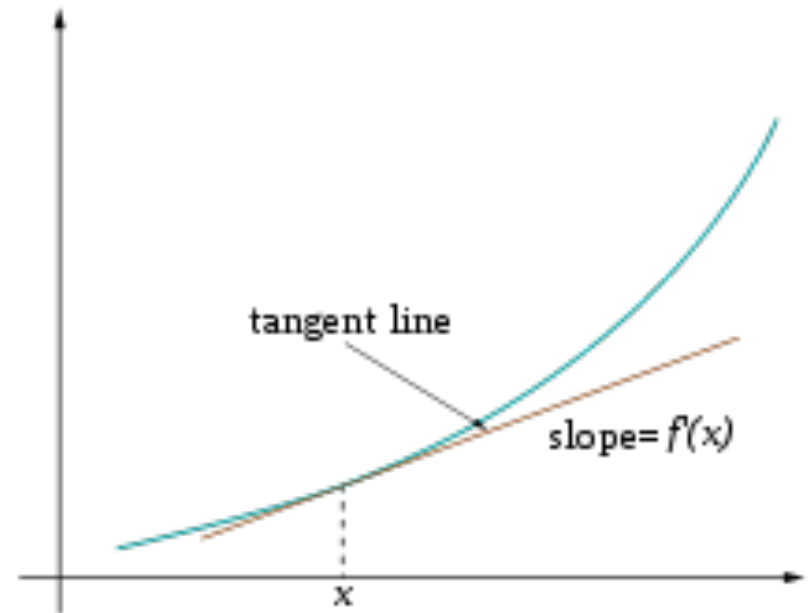
We will optimize (maximize or minimize) our objective/cost functions

For now: minimize → gradient descent

Refresher with trivial example: (from Wikipedia)

Find a local minimum of the function

$f(x)=x^4-3x^3+2$, with derivative $f'(x)=4x^3-9x^2$.



```
x_old = 0
x_new = 6 # The algorithm starts at x=6
eps = 0.01 # step size
precision = 0.00001

def f_derivative(x):
    return 4 * x**3 - 9 * x**2

while abs(x_new - x_old) > precision:
    x_old = x_new
    x_new = x_old - eps * f_derivative(x_old)

print("Local minimum occurs at", x_new)
```


Derivations of gradient

- Whiteboard (see video if you're not in class ;)
- Most basic Lego piece, speed will depend on participation
- Useful basics: $\frac{\partial \mathbf{x}^T \mathbf{a}}{\partial \mathbf{x}} = \frac{\partial \mathbf{a}^T \mathbf{x}}{\partial \mathbf{x}} = \mathbf{a}$
- Chain rule! If $y = f(u)$ and $u = g(x)$, i.e. $y=f(g(x))$, then:

$$\frac{dy}{dx} = \frac{dy}{du} \frac{du}{dx}$$

Whiteboard!

Approximations: PSet 1

- With large vocabularies this objective function is not scalable and would train too slowly! → Why?
- Idea: approximate the normalization or
- Define negative prediction that only samples a few words that do not appear in the context
- Similar to focusing on mostly positive correlations
- You will derive and implement this in Pset 1!

Linear Relationships in word2vec

These representations are *very good* at encoding dimensions of similarity!

- Analogies testing dimensions of similarity can be solved quite well just by doing vector subtraction in the embedding space

Syntactically

- $x_{apple} - x_{apples} \approx x_{car} - x_{cars} \approx x_{family} - x_{families}$

- Similarly for verb and adjective morphological forms

Semantically (Semeval 2012 task 2)

- $x_{shirt} - x_{clothing} \approx x_{chair} - x_{furniture}$

- $x_{king} - x_{man} \approx x_{queen} - x_{woman}$

Count based vs direct prediction

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

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

This is SVD based



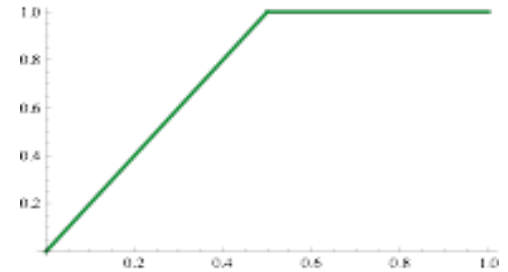
- NNLM, HLBL, RNN, Skip-gram/CBOW, (Bengio et al; Collobert & Weston; Huang et al; Mnih & Hinton; Mikolov et al; Mnih & Kavukcuoglu)

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

Combining the best of both worlds: GloVe

$$J = \frac{1}{2} \sum_{ij} f(P_{ij}) (w_i \cdot \tilde{w}_j - \log P_{ij})^2$$

$f \sim$



- 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



leptodactylidae



rana



eleutherodactylus

Word Analogies

Test for linear relationships, examined by Mikolov et al. (2014)

a:b :: c:?



$$d = \arg \max_x \frac{(w_b - w_a + w_c)^T w_x}{\|w_b - w_a + w_c\|}$$

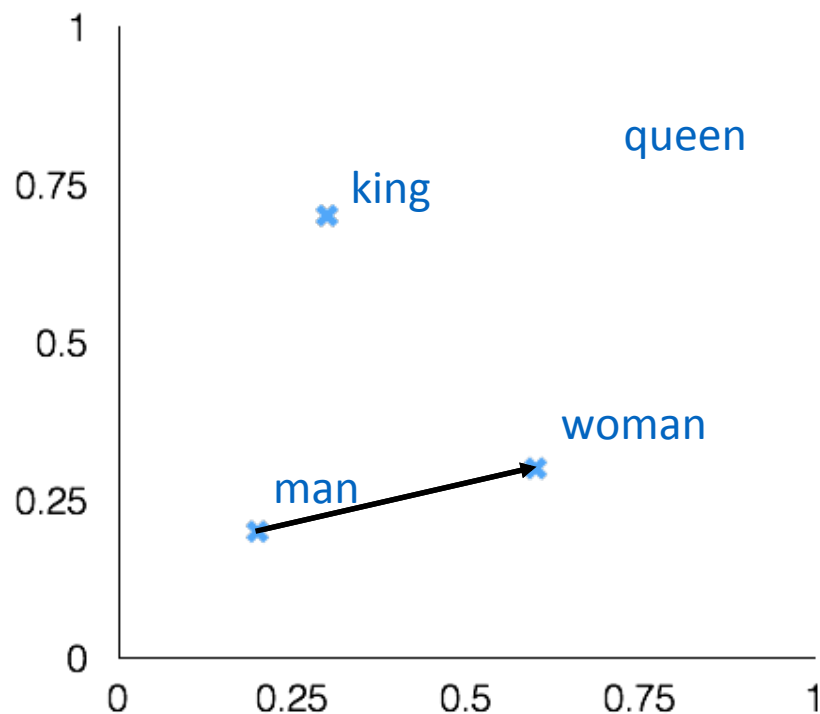
man:woman :: king:?

+ king [0.30 0.70]

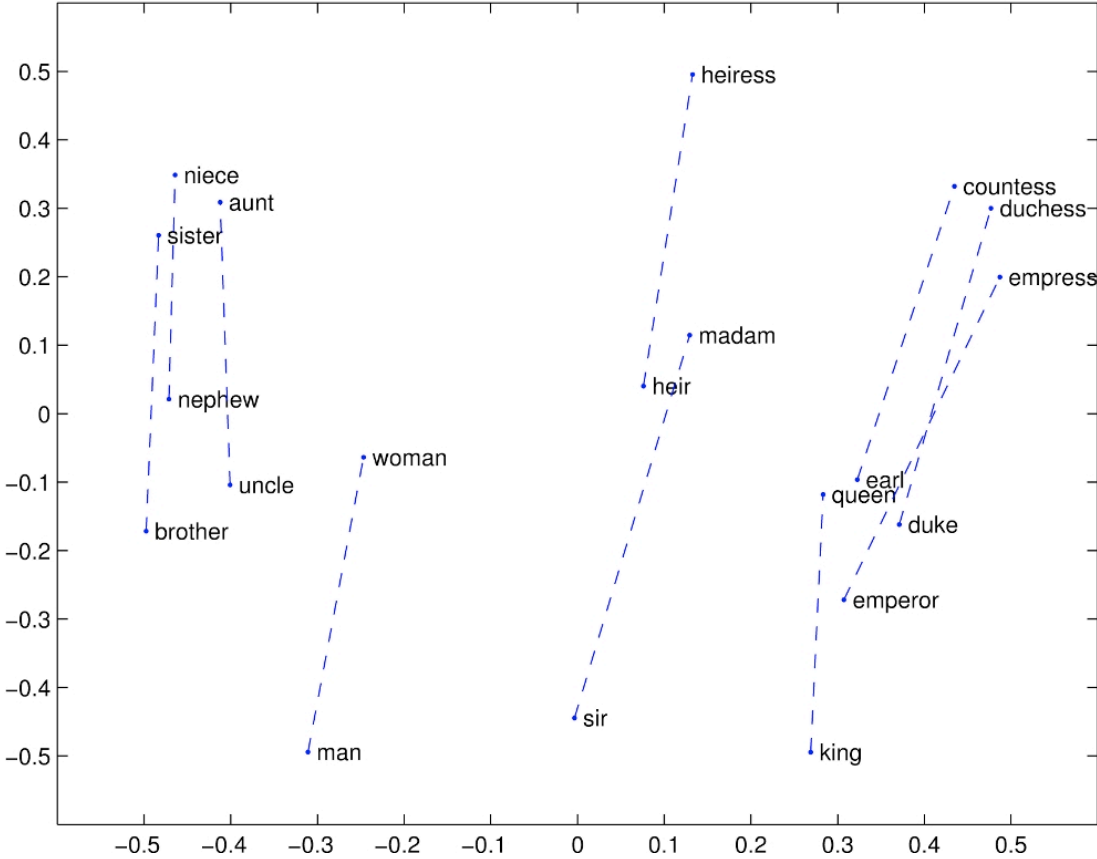
- man [0.20 0.20]

+ woman [0.60 0.30]

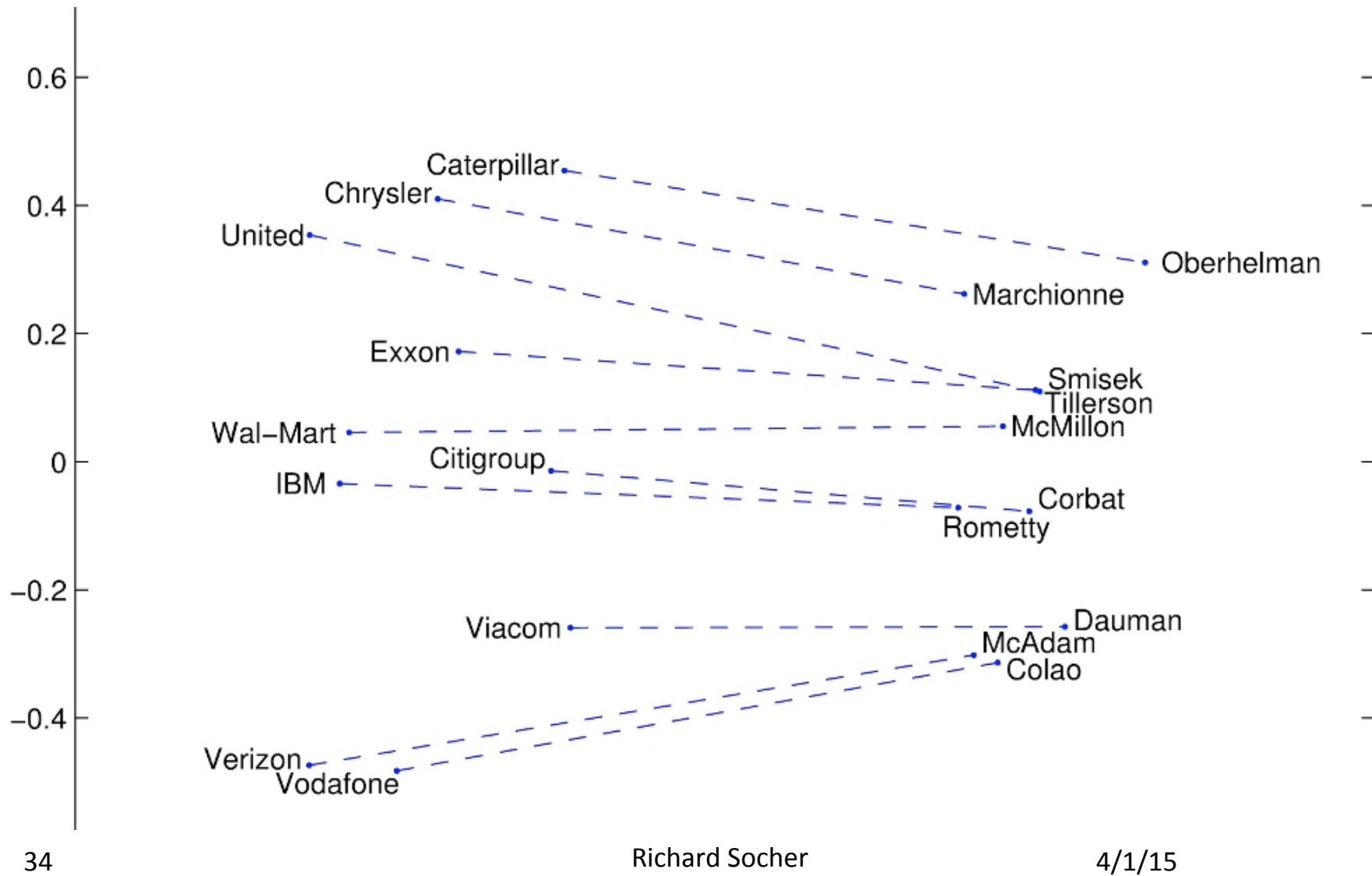
queen [0.70 0.80]



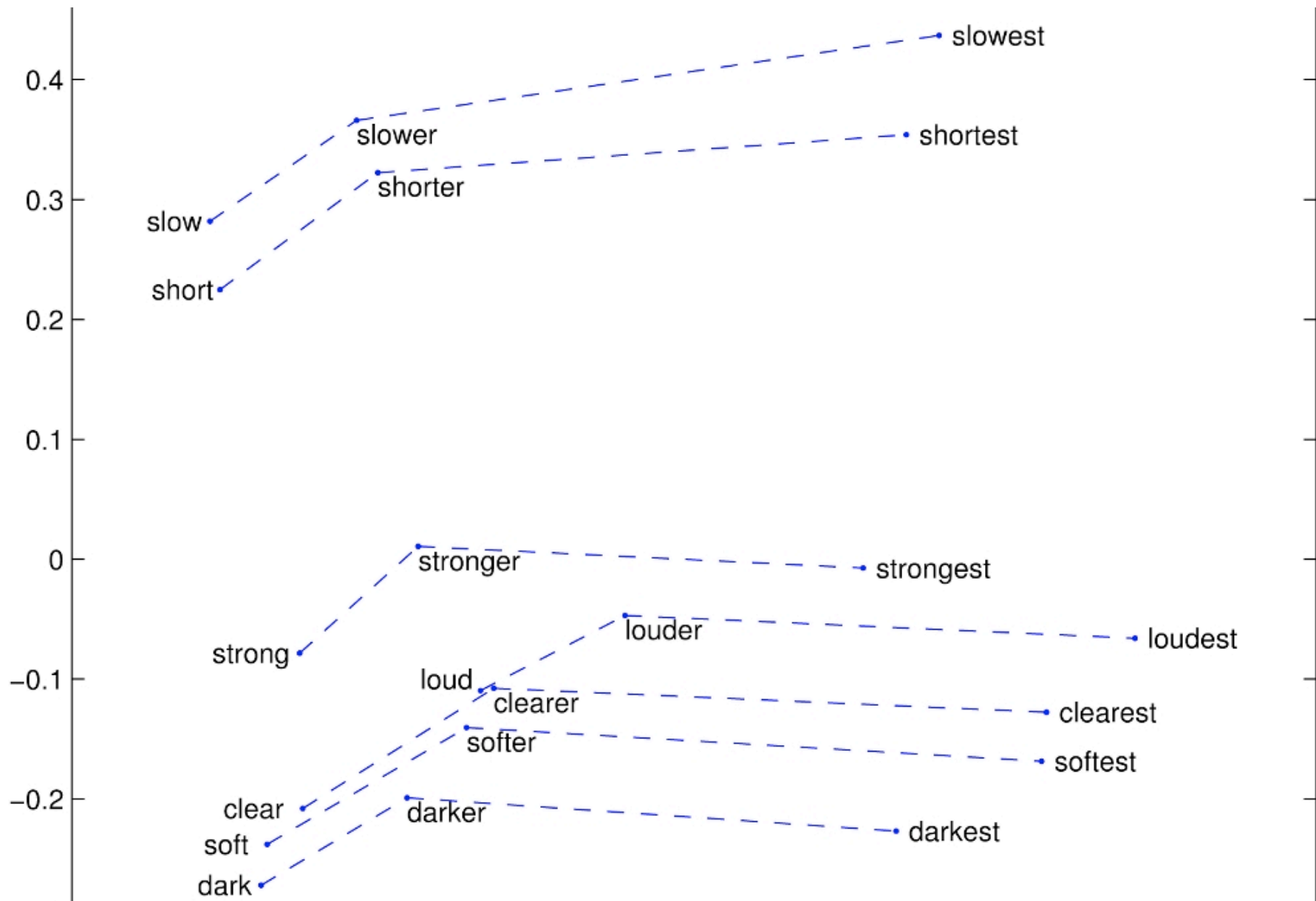
Glove Visualizations



Glove Visualizations: Company - CEO

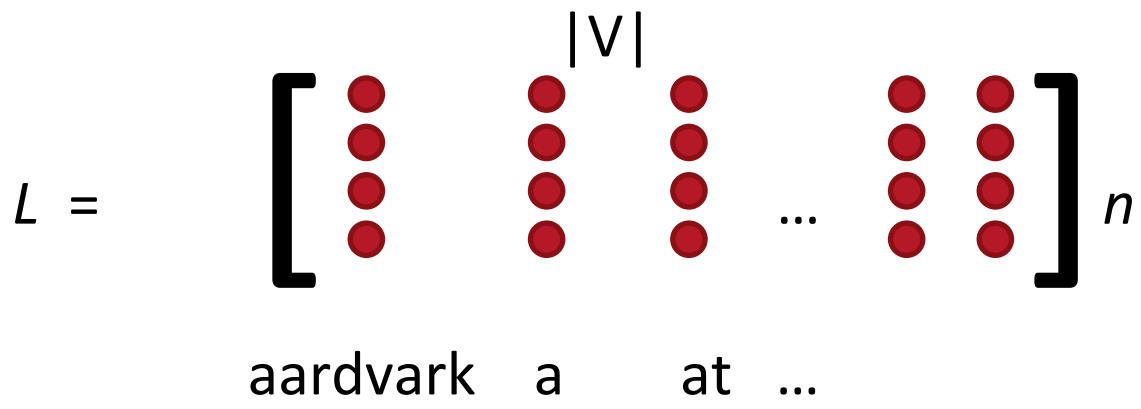


Glove Visualizations: Superlatives



Word embedding matrix

- Initialize most word vectors of future models with our “pre-trained” embedding matrix $L \in \mathbb{R}^{n \times |V|}$



- Also called a look-up table
 - Conceptually you get a word’s vector by left multiplying a one-hot vector e (of length $|V|$) by L : $x = Le$

Advantages of low dimensional word vectors

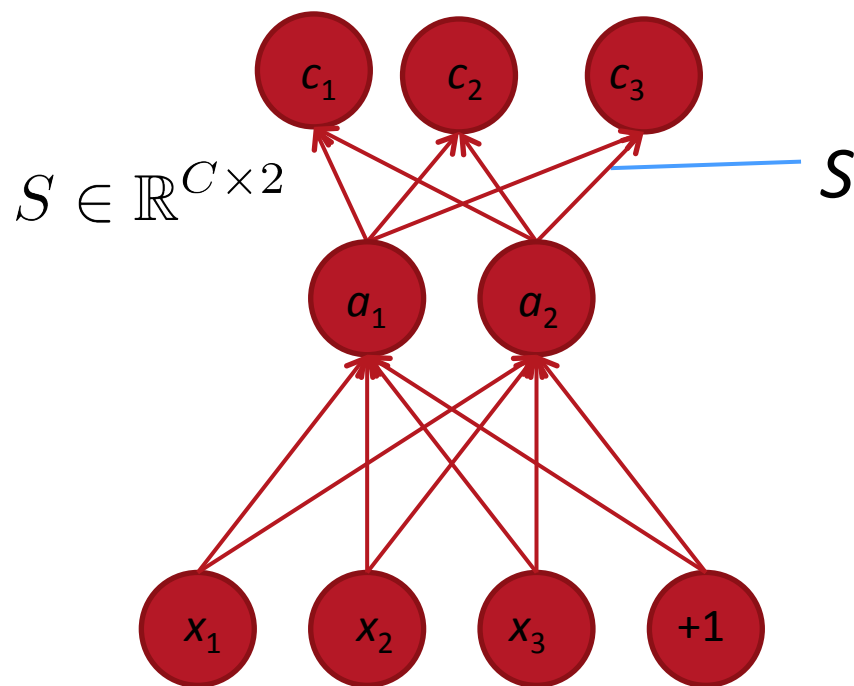
What is the major benefit of deep learned word vectors?

Ability to also propagate **any** information into them via neural networks (next lecture).

$$P(c|d, \lambda) = \frac{e^{\lambda^\top f(c,d)}}{\sum_{c'} e^{\lambda^\top f(c',d)}}$$



$$p(c|x) = \frac{\exp(S_c \cdot a)}{\sum_{c'} \exp(S_{c'} \cdot a)}$$



Advantages of low dimensional word vectors

- Word vectors will form the basis for all subsequent lectures.
- All our semantic representations will be vectors!
- We can compute compositional representations for longer phrases or sentences with them and solve lots of different tasks. → Next lecture!

Refresher: The simple word2vec model

- Main cost function J:

$$J(\theta) = \frac{1}{T} \sum_{t=1}^T \sum_{-c \leq j \leq c, j \neq 0} \log p(w_{t+j} | w_t)$$

- With probabilities defined as: $p(w_O | w_I) = \frac{\exp(v'_{w_O} \top v_{w_I})}{\sum_{w=1}^W \exp(v'_w \top v_{w_I})}$
- We derived the gradient for the internal vectors v_{w_I} (v_c on the board)

Calculating all gradients!

- We went through gradients for each center vector v in a window
- We also need gradients for external vectors v' (u on the board)
- Derive!

- Generally in each window we will compute updates for all parameters that are being used in that window.
- For example window size $c = 1$, sentence:
“I like learning .”
- First window computes gradients for:
 - internal vector v_{like} and external vectors v'_I and v'_{learning}
- Next window in that sentence?

Compute all vector gradients!

- We often define the set of ALL parameters in a model in terms of one long vector θ
- In our case with d -dimensional vectors and V many words:

$$\theta = \begin{bmatrix} v_{aardvark} \\ v_a \\ \vdots \\ v_{zebra} \\ v'_{aardvark} \\ v'_a \\ \vdots \\ v'_{zebra} \end{bmatrix} \in \mathbb{R}^{2dV}$$

Gradient Descent

- To minimize $J(\theta)$ over the full batch (the entire training data) would require us to compute gradients for all windows
- Updates would be for each element of θ :

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

- With step size α
- In matrix notation for all parameters:

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

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

Vanilla Gradient Descent Code

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

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

Intuition

- For a simple convex function over two parameters.
- Contour lines show levels of objective function
- See Whiteboard

Stochastic Gradient Descent

- But Corpus may have 40B tokens and windows
- You would wait a very long time before making a single update!
- Very bad idea for pretty much all neural nets!
- Instead: We will update parameters after each window t
→ Stochastic gradient descent (SGD)

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

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

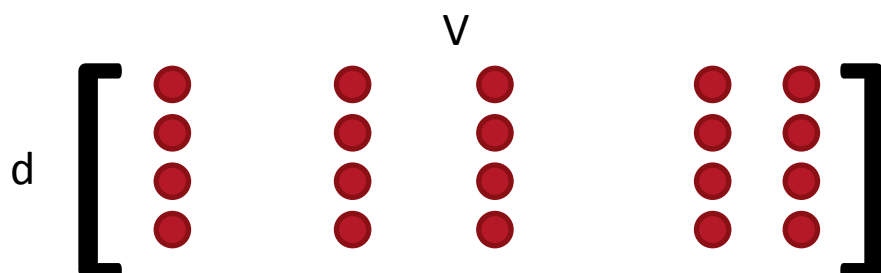
Stochastic gradients with word vectors!

- But in each window, we only have at most $2c - 1$ words, so $\nabla_{\theta} J_t(\theta)$ is very sparse!

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

Stochastic gradients with word vectors!

- We may as well only update the word vectors that actually appear!
- Solution: either keep around hash for word vectors or only update certain columns of full embedding matrix L and L'



- Important if you have millions of word vectors and do distributed computing to not have to send gigantic updates around.

Approximations: PSet 1

- The normalization factor is too computationally expensive

$$p(w_O | w_I) = \frac{\exp \left(v'_{w_O} \top v_{w_I} \right)}{\sum_{w=1}^W \exp \left(v'_w \top v_{w_I} \right)}$$

- Hence, in PSet1 you will implement the skip-gram model
- Main idea: train binary logistic regressions for a true pair (center word and word in its context window) and a couple of random pairs (the center word with a random word)

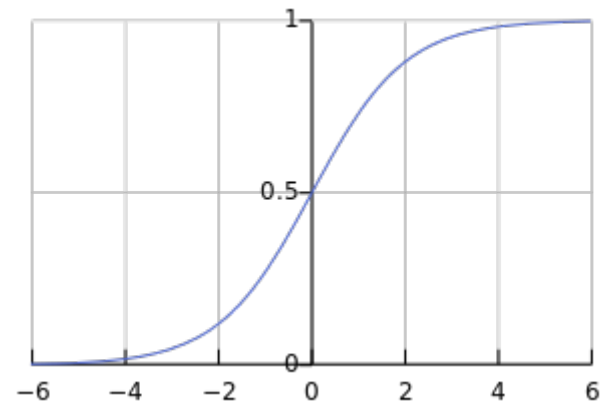
PSet 1: The skip-gram model and negative sampling

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

$$\log \sigma(v'_{w_O} \top v_{w_I}) + \sum_{i=1}^k \mathbb{E}_{w_i \sim P_n(w)} \left[\log \sigma(-v'_{w_i} \top v_{w_I}) \right]$$

- Where k is the number of negative samples and we use,

- The sigmoid function! $\sigma(x) = \frac{1}{1+e^{-x}}$
(we'll become good friends soon)
- So we maximize the probability of two words co-occurring in first log
→



PSet 1: The skip-gram model and negative sampling

- Slightly clearer notation:

$$\log \sigma (v_{wI}^T v'_{wO}) + \sum_{i \sim P_n(w)} \log \sigma (-v_{wI}^T v'_{wi})$$

- Max. probability that real outside word appears, minimize prob. that random words appear around center word
- $P_n = U(w)^{3/4}/Z$,
the unigram distribution $U(w)$ raised to the 3/4rd power
(We provide this function in the starter code).
- The power makes less frequent words be sampled more often

PSet 1: The continuous bag of words model

- Main idea for continuous bag of words (CBOW): Predict center word from sum of surrounding word vectors instead of predicting surrounding single words from center word as in skip-gram model

- To make PSet slightly easier:

The implementation for the CBOW model is not required and for bonus points!

What to do with the two sets of vectors?

- We end up with L and L' from all the vectors v and v'
- Both capture similar co-occurrence information. It turns out, the best solution is to simply sum them up:

$$L_{\text{final}} = L + L'$$

- One of many hyperparameters explored in *GloVe: Global Vectors for Word Representation* (Pennington et al. (2014))

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 one subsystem with another improves accuracy → Winning!

Intrinsic word vector evaluation

- Word Vector Analogies: Syntactic and Semantic

a:b :: c:?



$$d = \arg \max_x \frac{(w_b - w_a + w_c)^T w_x}{\|w_b - w_a + w_c\|}$$

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?

Intrinsic word vector evaluation

- Word Vector Analogies: Syntactic and **Semantic** examples from <http://code.google.com/p/word2vec/source/browse/trunk/questions-words.txt>

: city-in-state

Chicago Illinois Houston Texas

Chicago Illinois Philadelphia Pennsylvania

Chicago Illinois Phoenix Arizona

Chicago Illinois Dallas Texas

Chicago Illinois Jacksonville Florida

Chicago Illinois Indianapolis Indiana

Chicago Illinois Austin Texas

Chicago Illinois Detroit Michigan

Chicago Illinois Memphis Tennessee

Chicago Illinois Boston Massachusetts

problem: different cities
may have same name

Intrinsic word vector evaluation

- Word Vector Analogies: Syntactic and **Semantic** examples from

: capital-world

problem: can change

Abuja Nigeria Accra Ghana

Abuja Nigeria Algiers Algeria

Abuja Nigeria Amman Jordan

Abuja Nigeria Ankara Turkey

Abuja Nigeria Antananarivo Madagascar

Abuja Nigeria Apia Samoa

Abuja Nigeria Ashgabat Turkmenistan

Abuja Nigeria Asmara Eritrea

Abuja Nigeria Astana Kazakhstan

Intrinsic word vector evaluation

- Word Vector Analogies: **Syntactic** and Semantic examples from

: gram4-superlative

bad worst big biggest

bad worst bright brightest

bad worst cold coldest

bad worst cool coolest

bad worst dark darkest

bad worst easy easiest

bad worst fast fastest

bad worst good best

bad worst great greatest

Intrinsic word vector evaluation

- Word Vector Analogies: **Syntactic** and Semantic examples from

: gram7-past-tense

dancing danced decreasing decreased

dancing danced describing described

dancing danced enhancing enhanced

dancing danced falling fell

dancing danced feeding fed

dancing danced flying flew

dancing danced generating generated

dancing danced going went

dancing danced hiding hid

dancing danced hitting hit

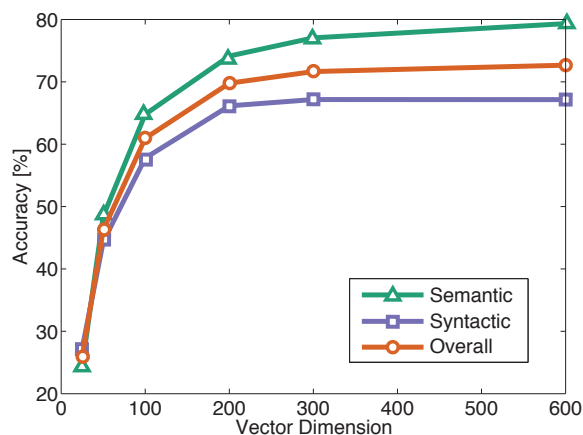
Analogy evaluation and hyperparameters

- Most careful analysis so far: Glove word vectors (which also capture cooccurrence counts but more directly so than skip-gram)

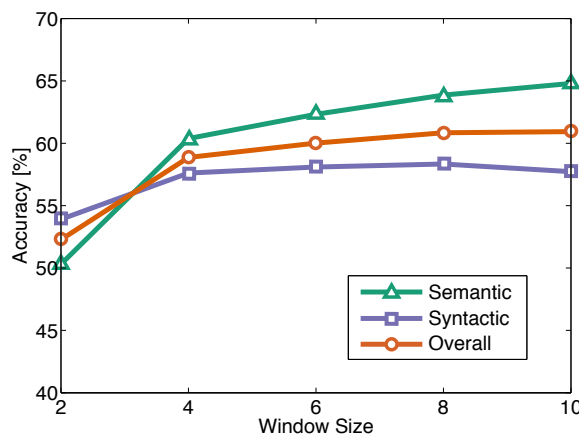
Model	Dim.	Size	Sem.	Syn.	Tot.
ivLBL	100	1.5B	55.9	50.1	53.2
HPCA	100	1.6B	4.2	16.4	10.8
GloVe	100	1.6B	<u>67.5</u>	<u>54.3</u>	<u>60.3</u>
SG	300	1B	61	61	61
CBOW	300	1.6B	16.1	52.6	36.1
vLBL	300	1.5B	54.2	<u>64.8</u>	60.0
ivLBL	300	1.5B	65.2	63.0	64.0
GloVe	300	1.6B	<u>80.8</u>	61.5	<u>70.3</u>
SVD	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>
CBOW	1000	6B	57.3	68.9	63.7
SG	1000	6B	66.1	65.1	65.6
SVD-L	300	42B	38.4	58.2	49.2
GloVe	300	42B	<u>81.9</u>	<u>69.3</u>	<u>75.0</u>

Analogy evaluation and hyperparameters

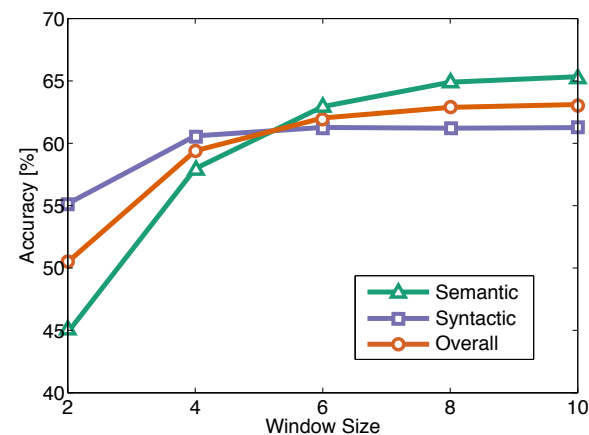
- Asymmetric context (only words to the left) are not as good



(a) Symmetric context



(b) Symmetric context

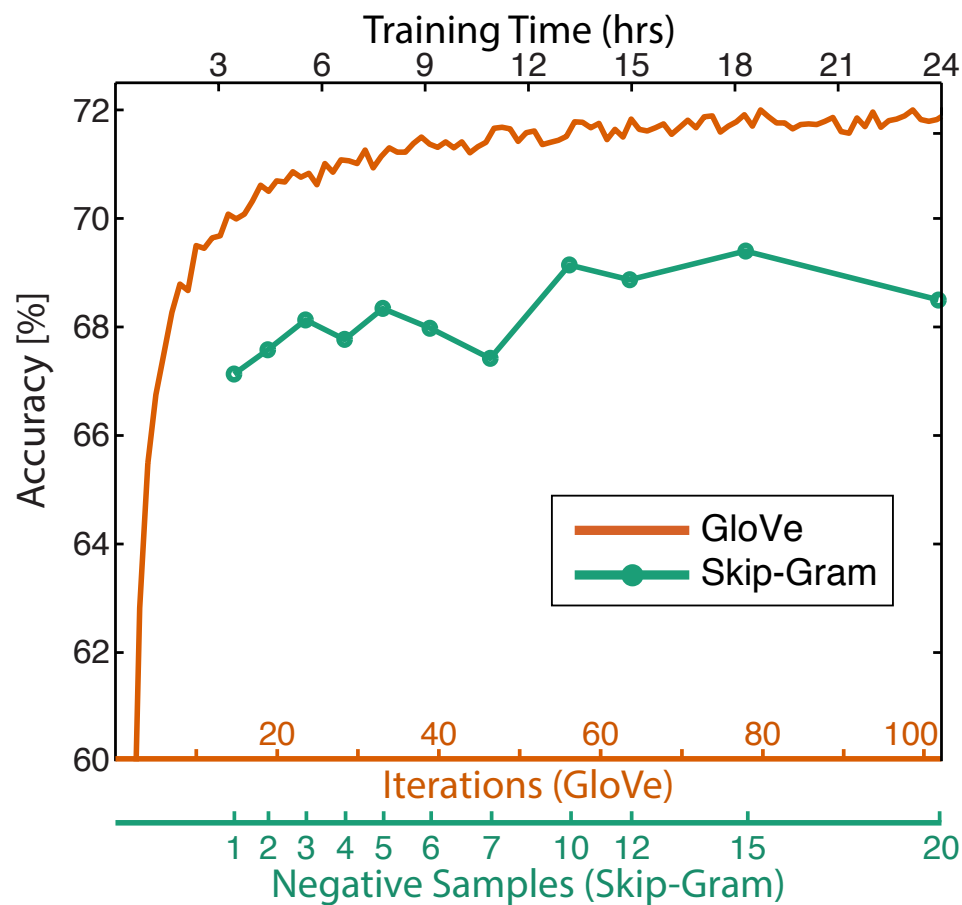


(c) Asymmetric context

- Best dimensions ~300, slight drop-off afterwards
- But this might be different for downstream tasks!
- Window size of 8 around each center word is good for Glove vectors

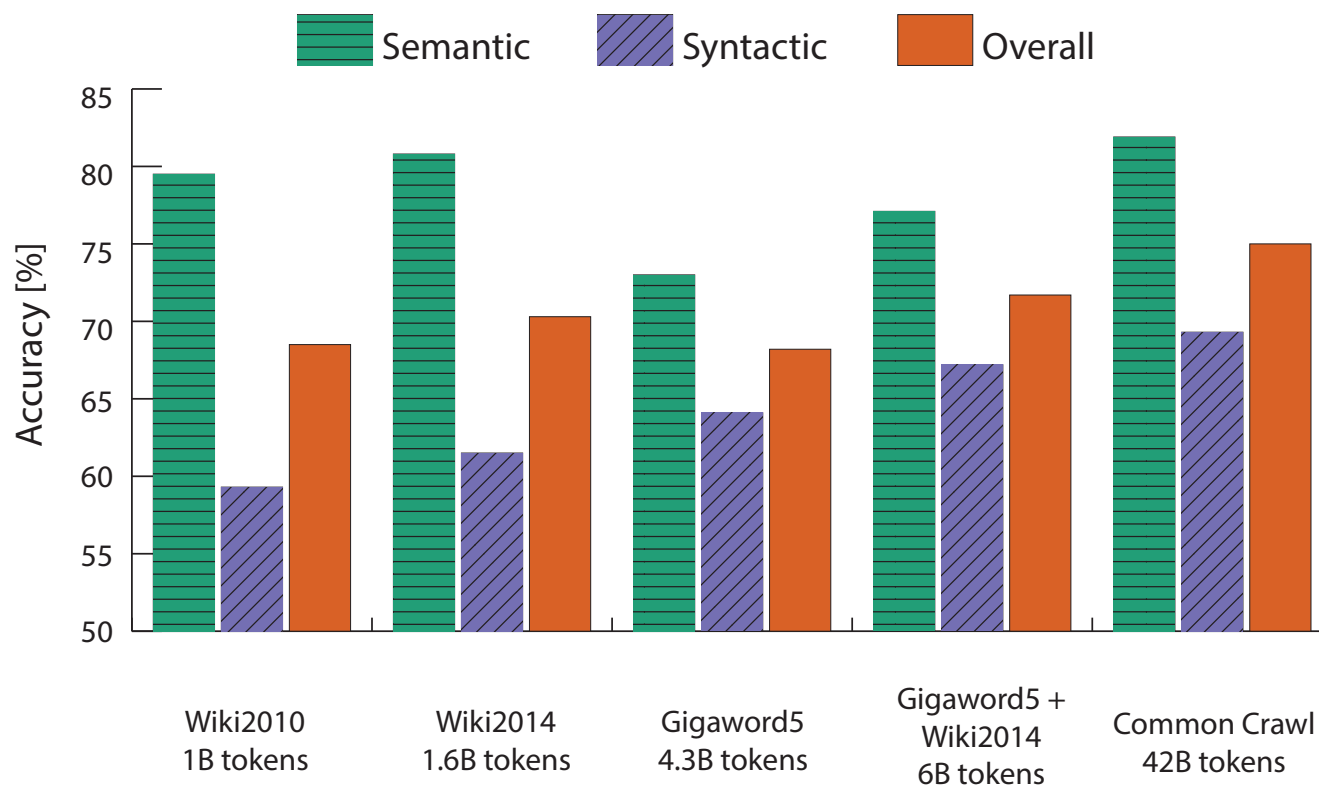
Analogy evaluation and hyperparameters

- More training time helps



Analogy evaluation and hyperparameters

- More data helps, Wikipedia is better than news text!



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	10.00
book	paper	7.46
computer	internet	7.58
plane	car	5.77
professor	doctor	6.62
stock	phone	1.62
stock	CD	1.31
stock	jaguar	0.92

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

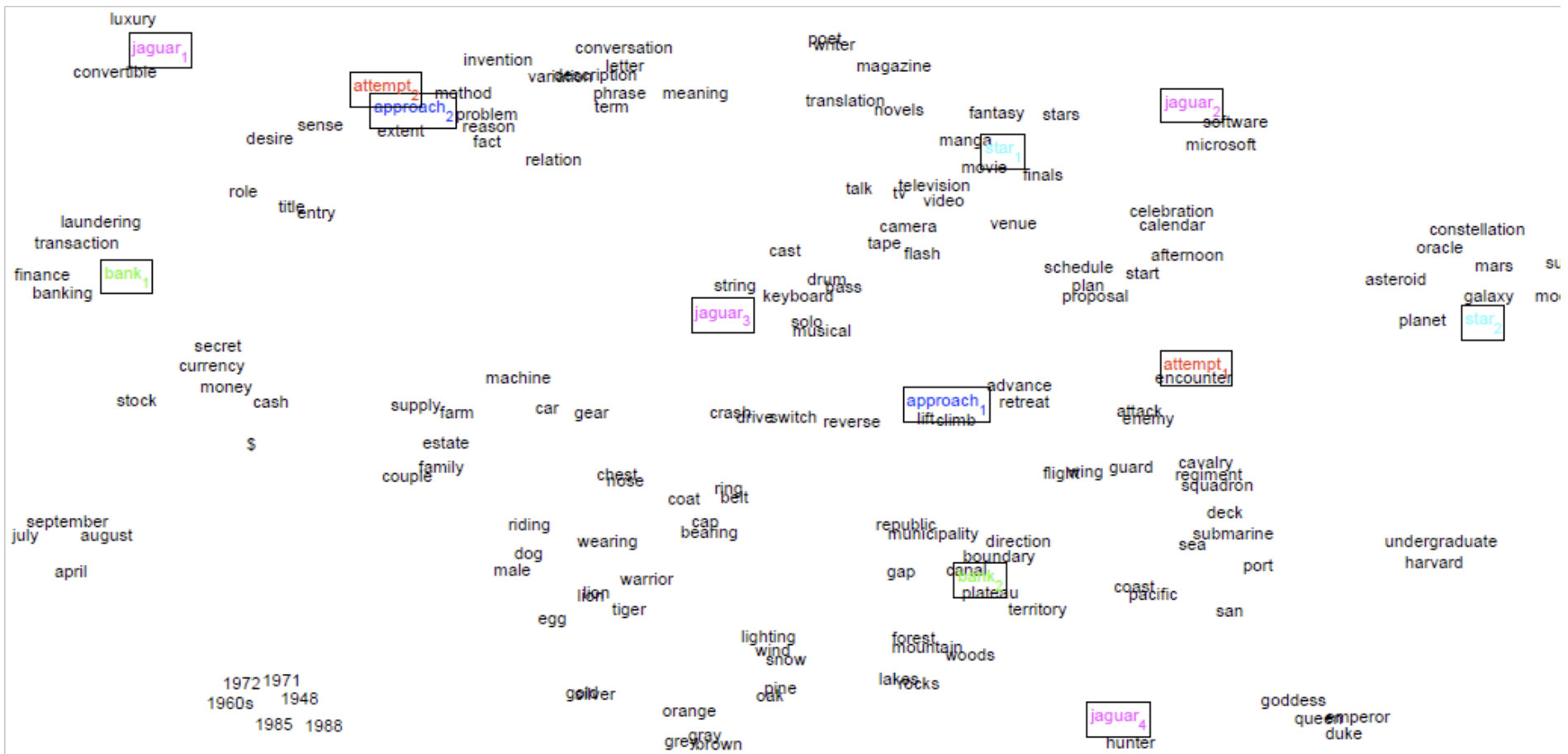
- Some ideas from Glove paper have been shown to improve skip-gram (SG) model also (e.g. sum both vectors)

But what about ambiguity?

- You may hope that one vector captures both kinds of information (run = verb and noun) but then vector is pulled in different directions
- Alternative described in: *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

But what about ambiguity?

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



Extrinsic word vector evaluation

- Extrinsic evaluation of word vectors: All subsequent tasks in this class
- One example where good word vectors should help directly: named entity recognition: finding 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

- Next: How to use word vectors in neural net models!

Simple single word classification

- What is the major benefit of deep learned word vectors?
 - Ability to also classify words accurately
 - Countries cluster together → classifying location words should be possible with word vectors
 - Incorporate **any** information into them other tasks
 - Project sentiment into words to find most positive/negative words in corpus

The softmax

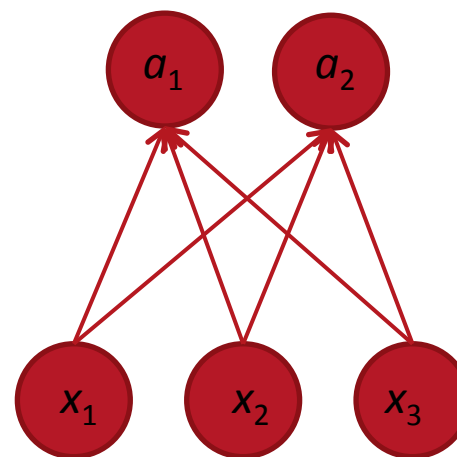
Logistic regression = Softmax classification on word vector x to obtain probability for class y :

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

where: $W \in \mathbb{R}^{C \times d}$

Generalizes >2 classes

(for just binary sigmoid unit would suffice as in skip-gram)



The softmax - details

- Terminology: Loss function = cost function = objective function
- Loss for softmax: Cross entropy
- To compute $p(y|x)$: first take the y 'th row of W and multiply that with row with x :

$$W_{y \cdot} x = \sum_{i=1}^d W_{yi} x_i = f_y$$

- Compute all f_c for $c=1, \dots, C$
- Normalize to obtain probability with softmax function:

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

The softmax and cross-entropy error

- The loss wants to maximize the probability of the correct class y
- Hence, we minimize the negative log probability of that class:

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

- As before: we sum up multiple cross entropy errors if we have multiple classifications in our total error function over the corpus (**more next lecture**)

Background: The Cross entropy error

- Assuming a ground truth (or gold or target) probability distribution that is 1 at the right class and 0 everywhere else: $p = [0, \dots, 0, 1, 0, \dots, 0]$ and our computed probability is q , then the cross entropy is:

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

- Because of one-hot p , the only term left is the negative probability of the true class
- Cross-entropy can be re-written in terms of the entropy and Kullback-Leibler divergence between the two distributions:

$$H(p, q) = H(p) + D_{KL}(p||q)$$

The KL divergence

- Cross entropy: $H(p, q) = H(p) + D_{KL}(p||q)$
- Because p is zero in our case (and even if it wasn't it would be fixed and have no contribution to gradient), to minimize this is equal to minimizing the KL divergence
- The KL divergence is **not a distance** but a non-symmetric measure of the difference between two probability distributions p and q

$$D_{KL}(p||q) = \sum_{c=1}^C p(c) \log \frac{p(c)}{q(c)}$$

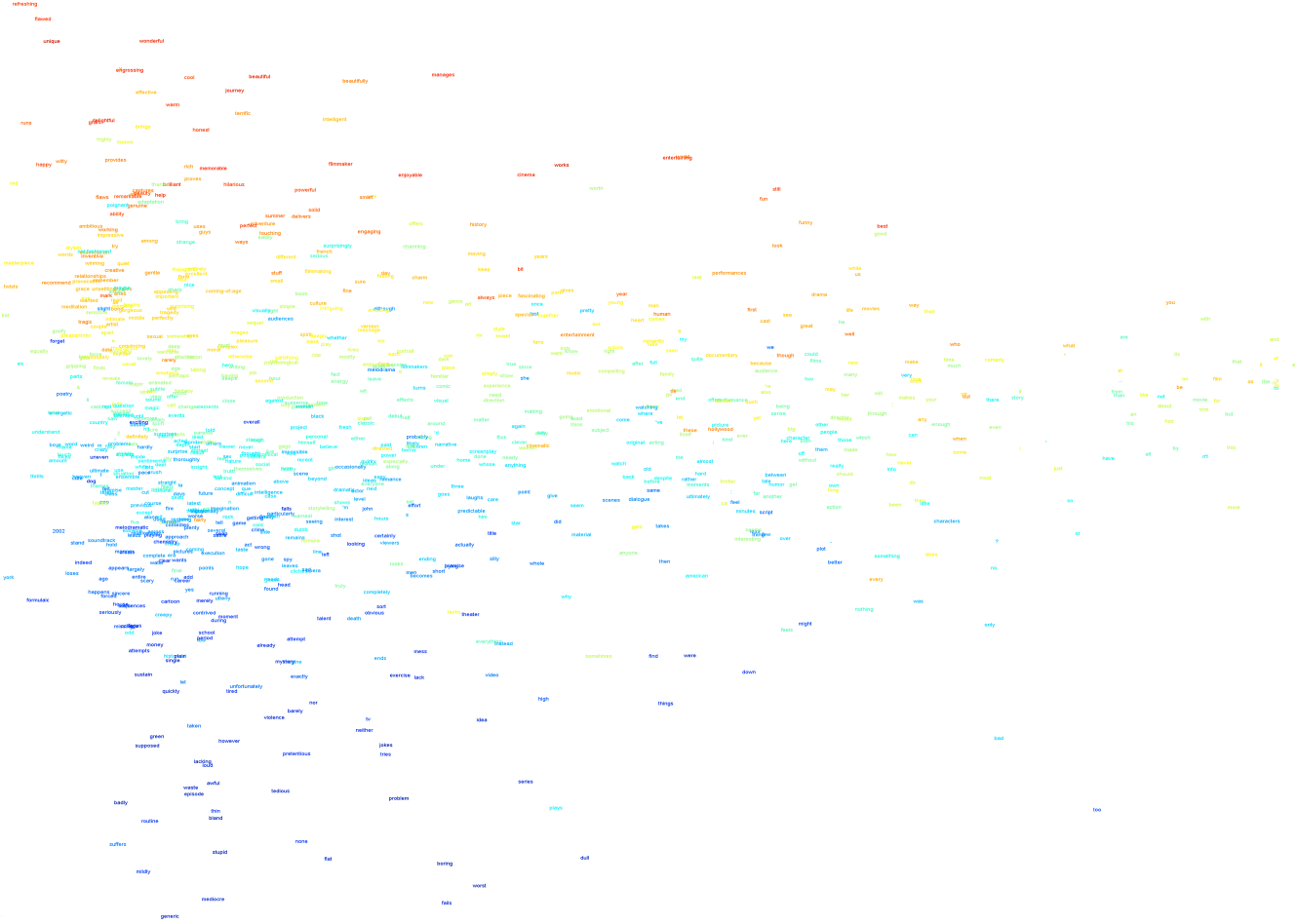
PSet 1

- Derive the gradient of the cross entropy error with respect to the input word vector x and the matrix W

Simple single word classification

- Example: Sentiment
- Two options: train only *softmax* weights W and fix word vectors or also train word vectors
- Question: What are the advantages and disadvantages of training the word vectors?
- Pro: better fit on training data
- Con: Worse generalization because the words move in the vector space

Visualization of sentiment trained word vectors



Next level up: Window classification

- Single word classification has no context!
- Let's add context by taking in windows and classifying the center word of that window!
- Possible: Softmax and cross entropy error or **max-margin loss**
- Next class!

References

Overview Today:

- General classification background
- Updating word vectors for classification
- Window classification & cross entropy error derivation tips
- A single layer neural network!
- (Max-Margin loss and **backprop**)

Refresher: Classification setup and notation

- Generally we have a training dataset consisting of samples

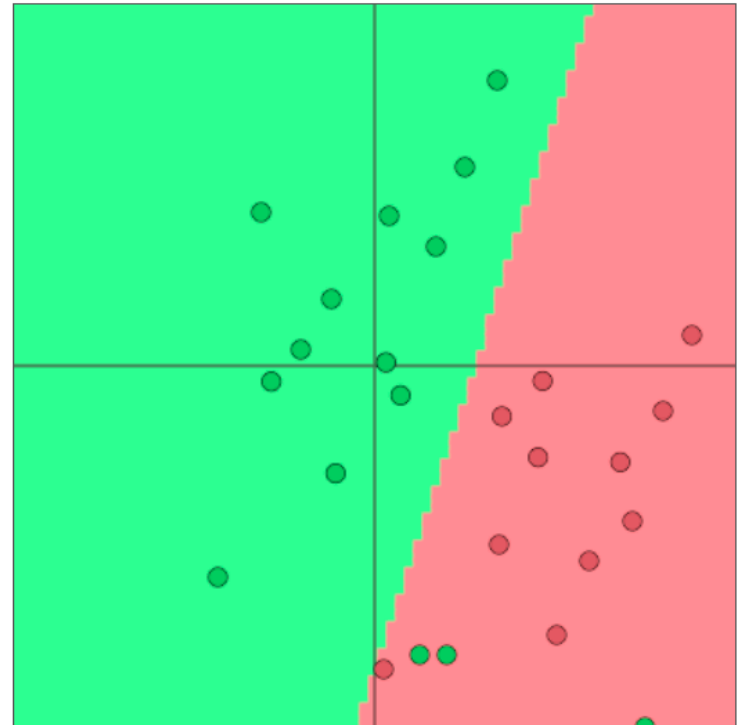
$$\{x_i, y_i\}_{i=1}^N$$

- x_i - inputs, e.g. words (indices or vectors!), context windows, sentences, documents, etc.
- y_i - labels we try to predict, e.g. sentiment, other words, named entities (loc., org. per.), buy/sell decision, later: multi-word sequences

Classification intuition

- Training data: $\{x_i, y_i\}_{i=1}^N$
- Simple illustration case:
 - Fixed 2d word vectors to classify
 - Using logistic regression
 - \rightarrow linear decision boundary \rightarrow
- General ML: assume x is fixed and only train logistic regression weights W and only modify the decision boundary

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



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

Classification notation

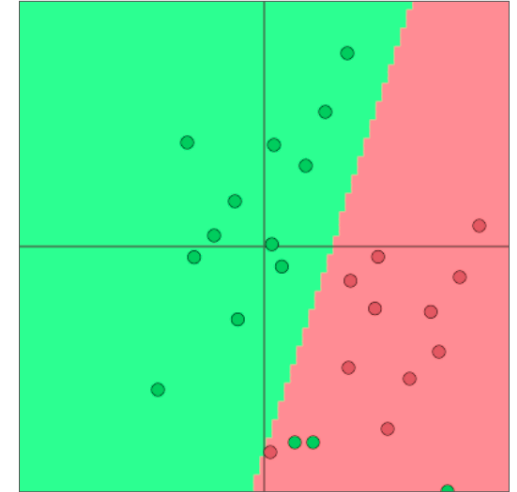
- General ML: only train logistic regression weights and hence only modify the decision boundary
- Loss function over dataset $\{x_i, y_i\}_{i=1}^N$

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

- Where for each data pair (x_i, y_i) :
- We can write f in matrix notation and index elements of it based on class:

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

$$f = Wx$$

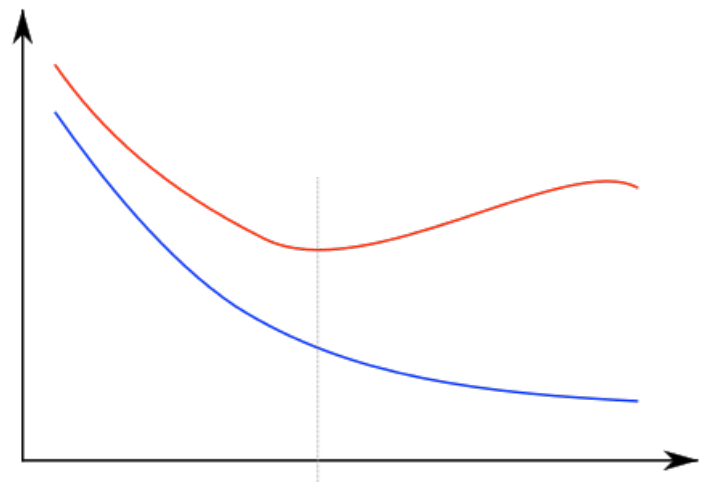


Classification: Regularization!

- Really full loss function over any dataset includes **regularization** over all parameters θ :

$$J(\theta) = \frac{1}{N} \sum_{i=1}^N -\log \left(\frac{e^{f_{y_i}}}{\sum_{c=1}^C e^{f_c}} \right) + \lambda \sum_k \theta_k^2$$

- Regularization will prevent overfitting when we have a lot of features (or later a very powerful/deep model)
 - x-axis: more powerful model or more training iterations
 - Blue: training error, red: test error



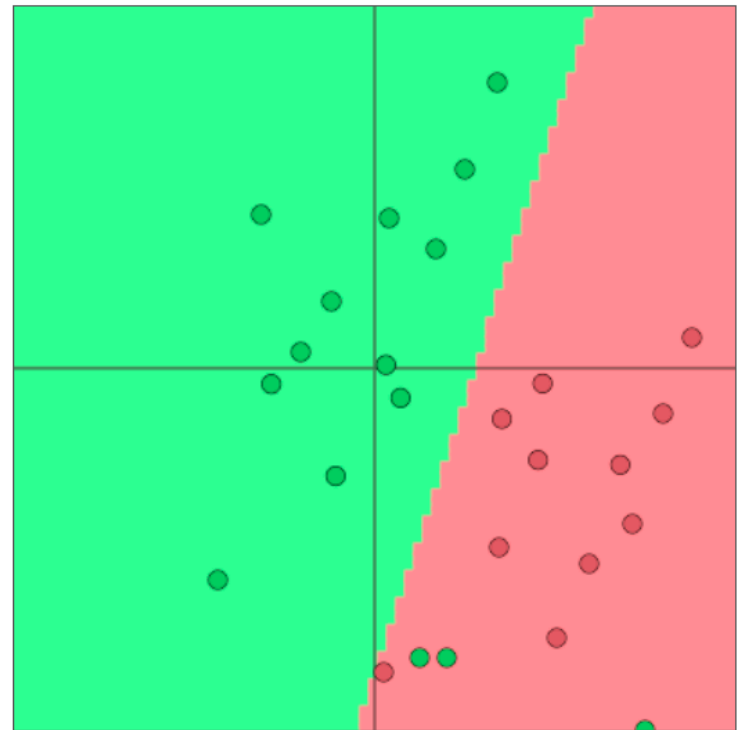
Classification difference with word vectors

- For general machine learning θ usually only consists of columns of W :

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

- So we only update the decision boundary

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



Visualizations with ConvNetJS by Karpathy

Classification difference with word vectors

- For general ML θ usually only consists of columns of W
- Additionally common in deep learning:
 - Learn both W and word vectors x

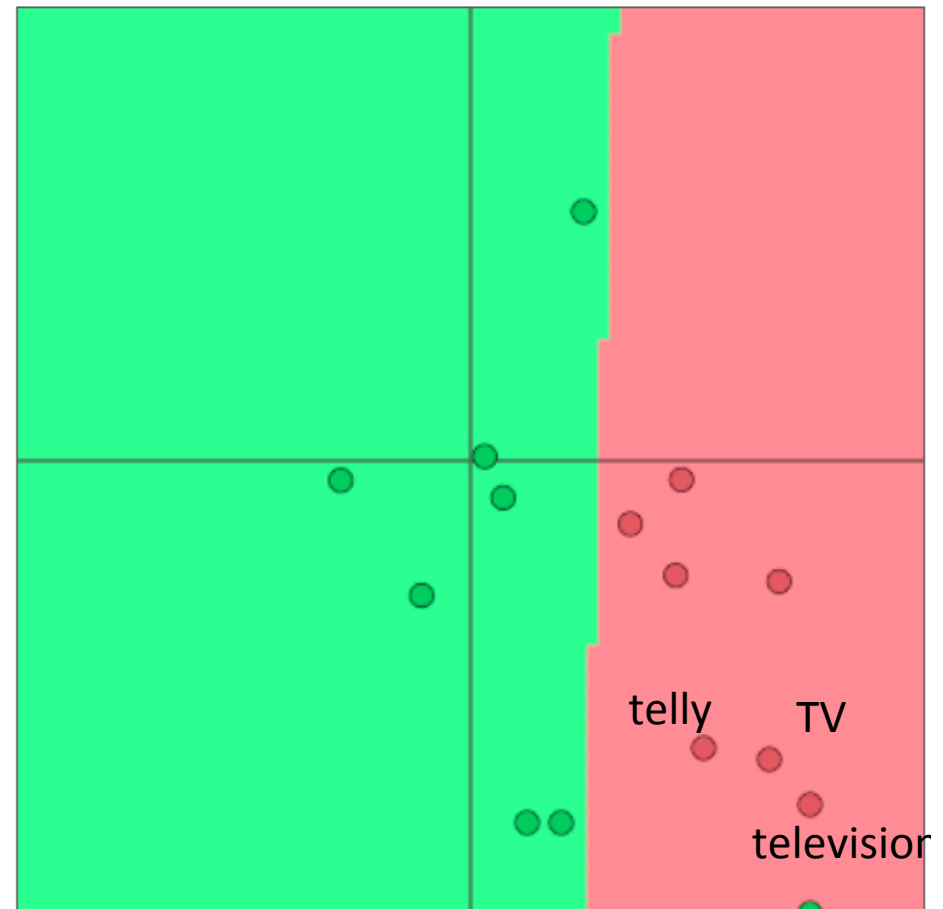
$$\nabla_{\theta} J(\theta) = \begin{bmatrix} \nabla_{W_{.1}} \\ \vdots \\ \nabla_{W_{.d}} \\ \nabla_{x_{aardvark}} \\ \vdots \\ \nabla_{x_{zebra}} \end{bmatrix} \in \mathbb{R}^{Cd+Vd}$$

Very large!

Overfitting Danger!

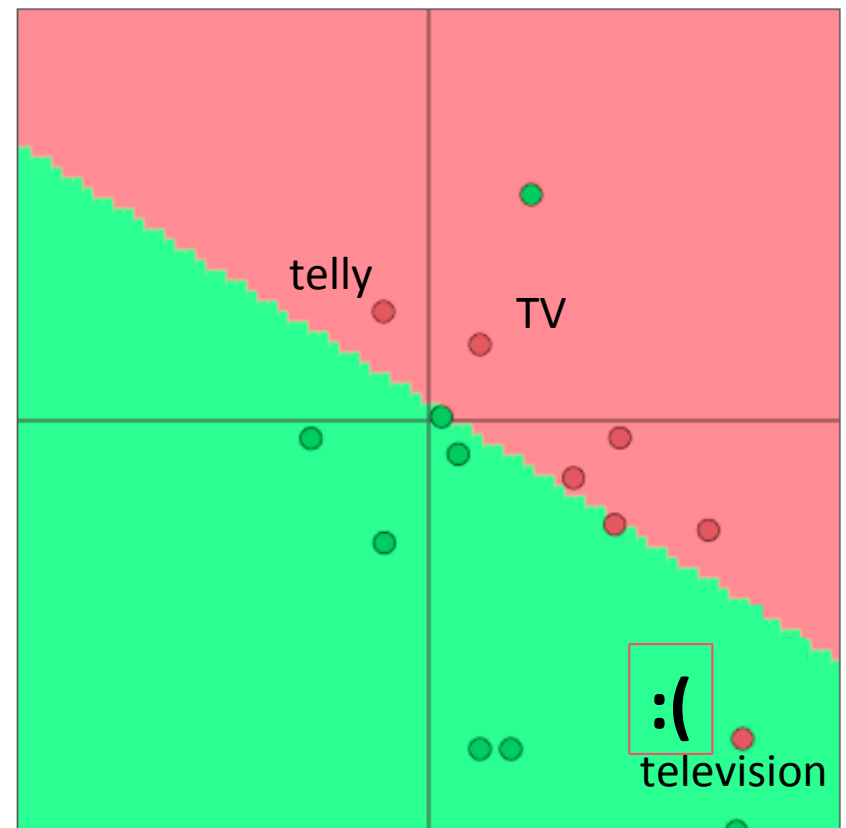
Loosing generalization by re-training word vectors

- Setting: Training logistic regression for movie review sentiment and in the training data we have the word
 - “TV” and “telly”
- In the testing data we have
 - “television”
- Originally they were all similar (from pre-training word vectors)
- What happens when we train the word vectors?



Loosing generalization by re-training word vectors

- What happens when we train the word vectors?
 - Those that are in the training data move around
 - Words from pre-training that do NOT appear in training stay
- Example:
 - In training data: “TV” and “telly”
 - In testing data only: “television”

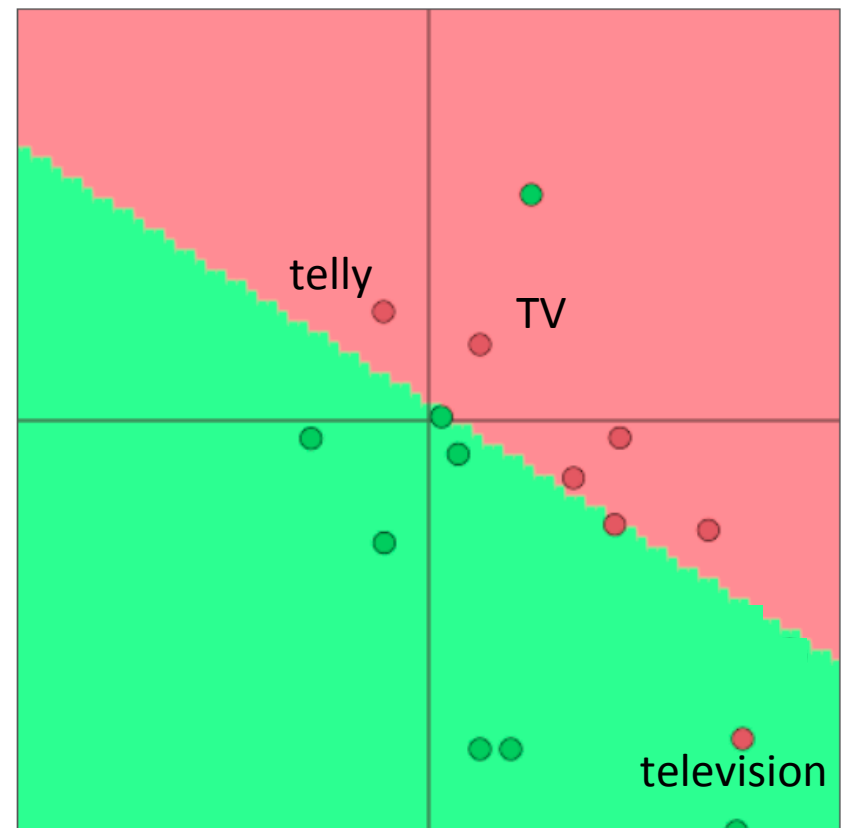


Loosing generalization by re-training word vectors

- Take home message:

If you only have a small training data set, don't train the word vectors.

If you have have a very large dataset, it may work better to train word vectors to the task.



Window classification

- Classifying single words is rarely done.
- Interesting problems like ambiguity arise in context!
- Example: auto-antonyms:
 - "To sanction" can mean "to permit" or "to punish."
 - "To seed" can mean "to place seeds" or "to remove seeds."
- Example: ambiguous named entities:
 - Paris → Paris Hilton vs Paris, France
 - Hathaway → Berkshire Hathaway, Anne Hathaway


Window classification

- Idea: Instead of classifying a single word, just classify a word together with its context window of neighboring words.
- For example named entity recognition into 4 classes:
 - Person, location, organization, none
- Many possibilities exist for classifying one word in context, e.g. averaging all the words in a window but that loses position information

Window classification

- Most commonly used technique to classify a word in a window
- Train classifier by assigning a label to a center word and concatenating all word vectors surrounding it.
- Example: Classify Paris in the context of this sentence with window length 2:

... museums in Paris are amazing ...



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

- Resulting vector $x_{\text{window}} = \boxed{x \in \mathbb{R}^{5d}}$, a column vector!

Simplest window classifier: Softmax

- With $x = x_{window}$ we can use the same softmax classifier as before

= predicted model output probability

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

- With cross entropy error as before:

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

- But how do you update the word vectors?

Updating concatenated word vectors

- Short answer: Just take derivatives as before
- Long answer: Let's go over the steps together (you'll have to fill in the details in PSet 1!)
- Define:
 - \hat{y} : softmax probability output vector (see previous slide)
 - t : target probability distribution (all 0's except at ground truth index of class y , where it's 1)
 - $f = Wx \in \mathbb{R}^C$ and $f_c = c$ 'th element of the f vector
- Hard, the first time, hence some tips now :)

Updating concatenated word vectors

- Tip 1: Carefully define your variables and keep track of their dimensionality! $f = f(x) = Wx \in \mathbb{R}^C$
 $\hat{y} \quad t \quad W \in \mathbb{R}^{C \times 5d}$
- Tip 2: **Know thy chain rule** and don't forget in which variables other variables are being used:

$$\frac{\partial}{\partial x} - \log \text{softmax}(f_y(x)) = \sum_{c=1}^C - \frac{\partial \log \text{softmax}(f_y(x))}{\partial f_c} \cdot \frac{\partial f_c(x)}{\partial x}$$

- Tip 3: For the softmax part of the derivative: First take the derivative wrt f_c when $c=y$ (the correct class), then take derivative wrt f_c when $c \neq y$ (all the incorrect classes)

Updating concatenated word vectors

- Tip 4: When you take derivative wrt one element of f , try to see if you can create a gradient in the end that includes all partial derivatives:

$$\hat{y} \quad t$$
$$f = f(x) = Wx \in \mathbb{R}^C$$

$$\frac{\partial}{\partial f} - \log \text{softmax}(f_y) = \begin{bmatrix} \hat{y}_1 \\ \vdots \\ \hat{y}_y - 1 \\ \vdots \\ \hat{y}_C \end{bmatrix}$$

- Tip 5: To later not go insane, think of your results in terms of vector operations and define new, single index-able vectors:

$$\frac{\partial}{\partial f} - \log \text{softmax}(f_y) = [\hat{y} - t] = \delta$$

Updating concatenated word vectors

- Tip 5: When you start with the chain rule, first use explicit sums and look at partial derivatives of e.g. x_i or W_{ij}

$$\hat{y} = t$$

$$f = f(x) = Wx \in \mathbb{R}^C$$

$$\sum_{c=1}^C \frac{\partial \log \text{softmax}(f_y(x))}{\partial f_c} \cdot \frac{\partial f_c(x)}{\partial x} = \sum_{c=1}^C \delta_c W_c.$$

- Tip 6: To clean it up for even more complex functions later: Know dimensionality of variables & simplify into matrix notation

$$\frac{\partial}{\partial x} - \log p(y|x) = \sum_{c=1}^C \delta_c W_c = W^T \delta$$

- Tip 7: Write this out in full sums if it's not clear!

Updating concatenated word vectors

- Tip 5: When you start with the chain rule, first use explicit sums and look at partial derivatives of e.g. x_i or W_{ij}

$$\hat{y} = t$$
$$f = f(x) = Wx \in \mathbb{R}^C$$

$$\sum_{c=1}^C \frac{\partial \log \text{softmax}(f_y(x))}{\partial f_c} \cdot \frac{\partial f_c(x)}{\partial x} = \sum_{c=1}^C \delta_c W_c.$$

- Tip 6: To clean it up for even more complex functions later: Know dimensionality of variables & simplify into matrix notation

$$\frac{\partial}{\partial x} - \log p(y|x) = \sum_{c=1}^C \delta_c W_c = W^T \delta$$

- Tip 7: Write this out in full sums if it's not clear!

Updating concatenated word vectors

- What is the dimensionality of

$$\frac{\partial}{\partial x} - \log p(y|x) = \sum_{c=1}^C \delta_c W_c = W^T \delta$$

- X is the entire window of 5 d -dimensional word vectors, so the derivative wrt to x has to have the same dimensionality:

$$\nabla_x J = W^T \delta \in \mathbb{R}^{5d}$$

Updating concatenated word vectors

- The gradient that arrives at and updates the word vectors can simply be split up for each word vector:

- Let $\nabla_x J = W^T \delta = \delta_{window}$

- With $x_{window} = [x_{museums} \quad x_{in} \quad x_{Paris} \quad x_{are} \quad x_{amazing}]$

- We have

$$\delta_{window} = \begin{bmatrix} \nabla_{x_{museums}} \\ \nabla_{x_{in}} \\ \nabla_{x_{Paris}} \\ \nabla_{x_{are}} \\ \nabla_{x_{amazing}} \end{bmatrix} \in \mathbb{R}^{5d}$$

Updating concatenated word vectors

- This will push word vectors into areas such they will be helpful in determining named entities.
- For example, the model can learn that seeing x_{i_n} as the word just before the center word is indicative for the center word to be a location

What's missing for training the window model?

- The gradient of J wrt the softmax weights W !
- Similar steps, write down partial wrt W_{ij} first!
- Then we have full

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

A note on matrix implementations

- There are two expensive operations in the softmax:
- The matrix multiplication $f = Wx$ and the exp
- A for loop is never as efficient when you implement it compared vs when you use a larger matrix multiplication that does the same mathematical operation!
- Example code →

A note on matrix implementations

- Looping over word vectors instead of concatenating them all into one large matrix and then multiplying the softmax weights with that matrix

```
from numpy import random
N = 500 # number of windows to classify
d = 300 # dimensionality of each window
C = 5 # number of classes
W = random.rand(C,d)
wordvectors_list = [random.rand(d,1) for i in range(N)]
wordvectors_one_matrix = random.rand(d,N)

%timeit [W.dot(wordvectors_list[i]) for i in range(N)]
%timeit W.dot(wordvectors_one_matrix)
```

- 1000 loops, best of 3: 639 μ s per loop
10000 loops, best of 3: 53.8 μ s per loop

A note on matrix implementations

```
from numpy import random
N = 500 # number of windows to classify
d = 300 # dimensionality of each window
C = 5 # number of classes
W = random.rand(C,d)
wordvectors_list = [random.rand(d,1) for i in range(N)]
wordvectors_one_matrix = random.rand(d,N)

%timeit [W.dot(wordvectors_list[i]) for i in range(N)]
%timeit W.dot(wordvectors_one_matrix)
```

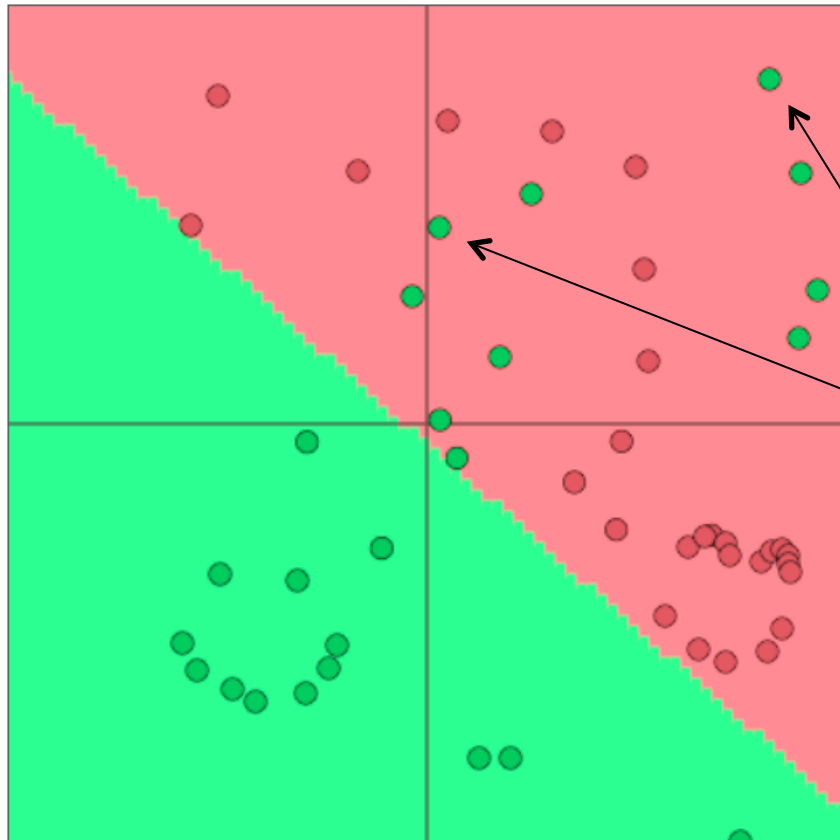
- Result of faster method is a $C \times N$ matrix:
 - Each column is an $f(x)$ in our notation (unnormalized class scores)
- Matrices are awesome!
- You should speed test your code a lot too

Softmax (= logistic regression) is not very powerful

- Softmax only gives linear decision boundaries in the original space.
- With little data that can be a good regularizer
- With more data it is very limiting!

Softmax (= logistic regression) is not very powerful

- Softmax only linear decision boundaries

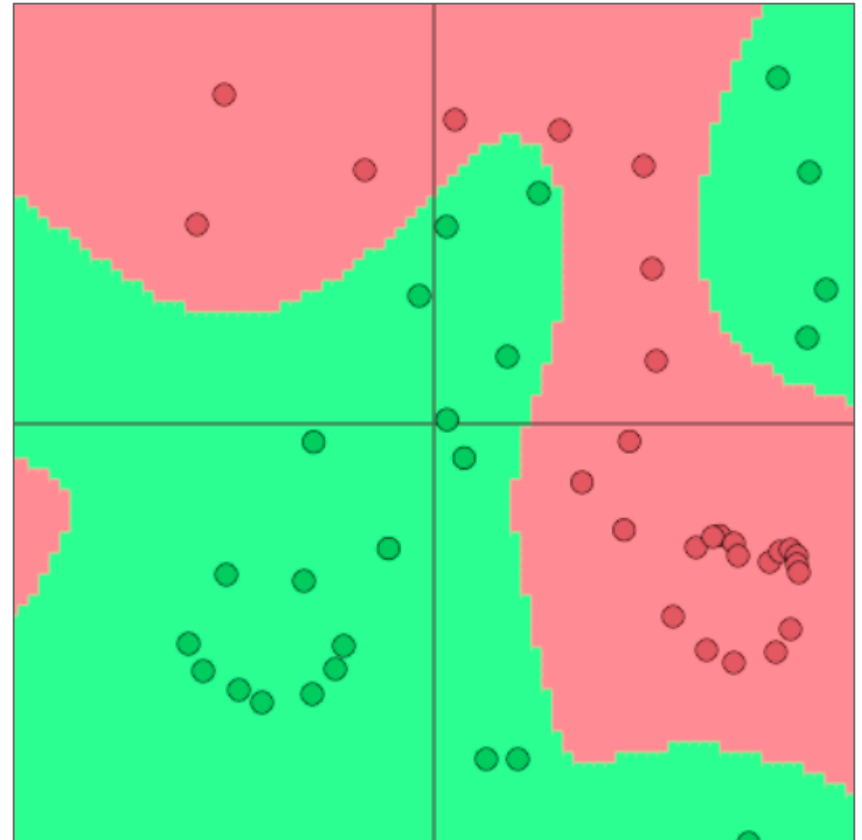
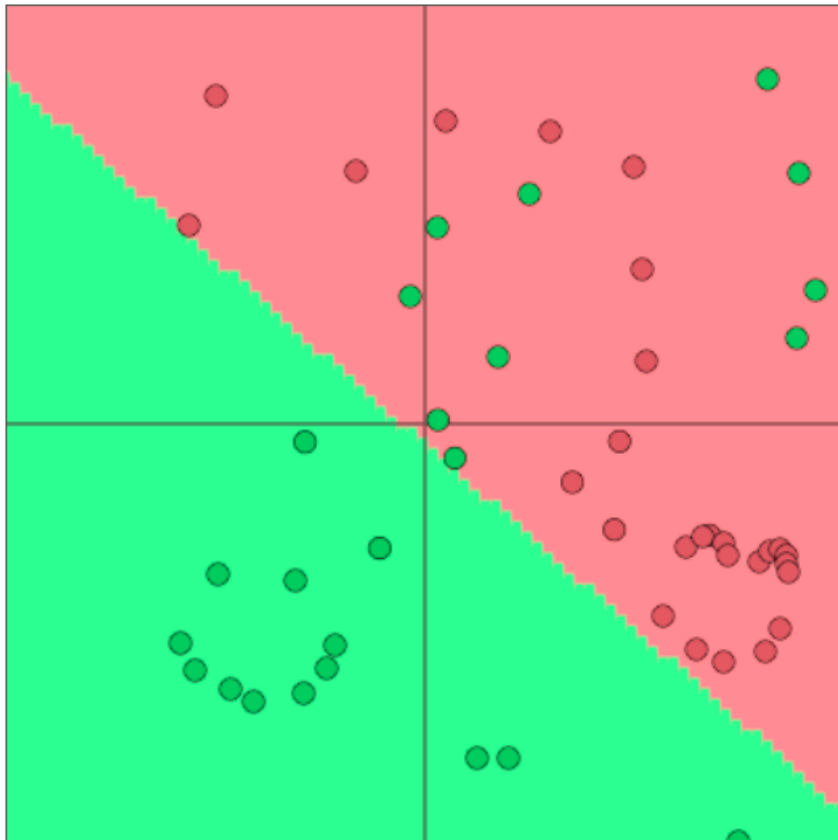


→ Lame when problem is complex

Wouldn't it be cool to get these correct?

Neural Nets for the Win!

- Neural networks can learn much more complex functions and nonlinear decision boundaries!



From logistic regression to neural nets

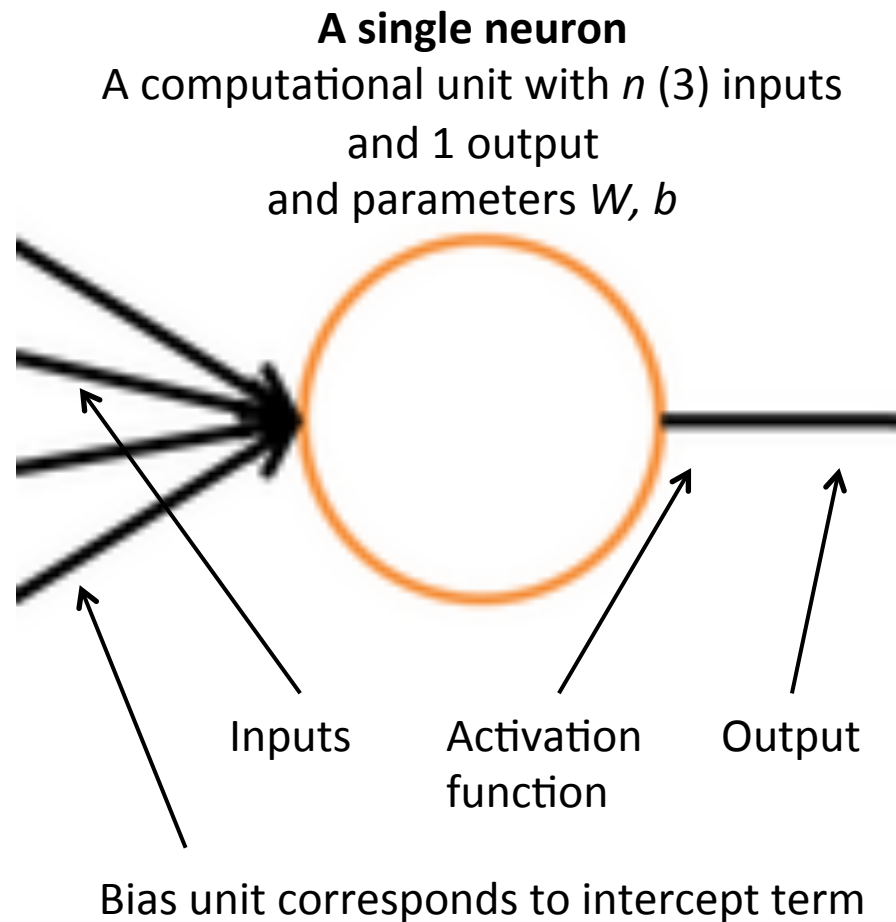
Demystifying neural networks

Neural networks come with their own terminological baggage

... just like SVMs

But if you understand how softmax models work

Then **you already understand** the operation of a basic neural network neuron!

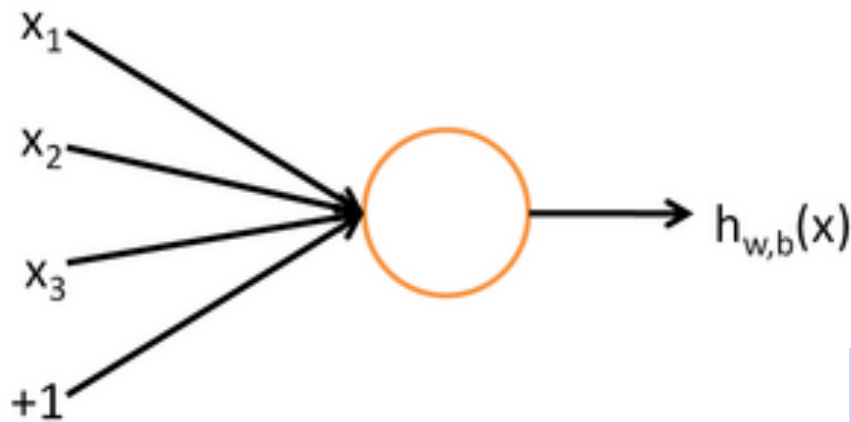
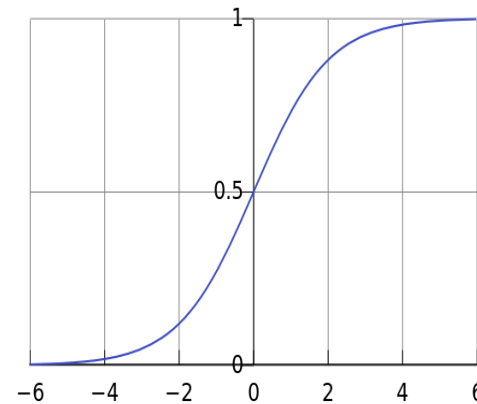


A neuron is essentially a binary logistic regression unit

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

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

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

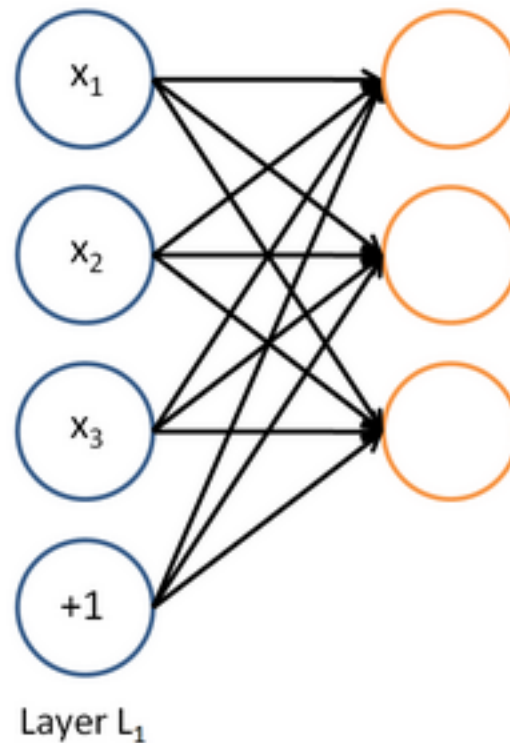


w, b are the parameters of this neuron i.e., this logistic regression model

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 ...

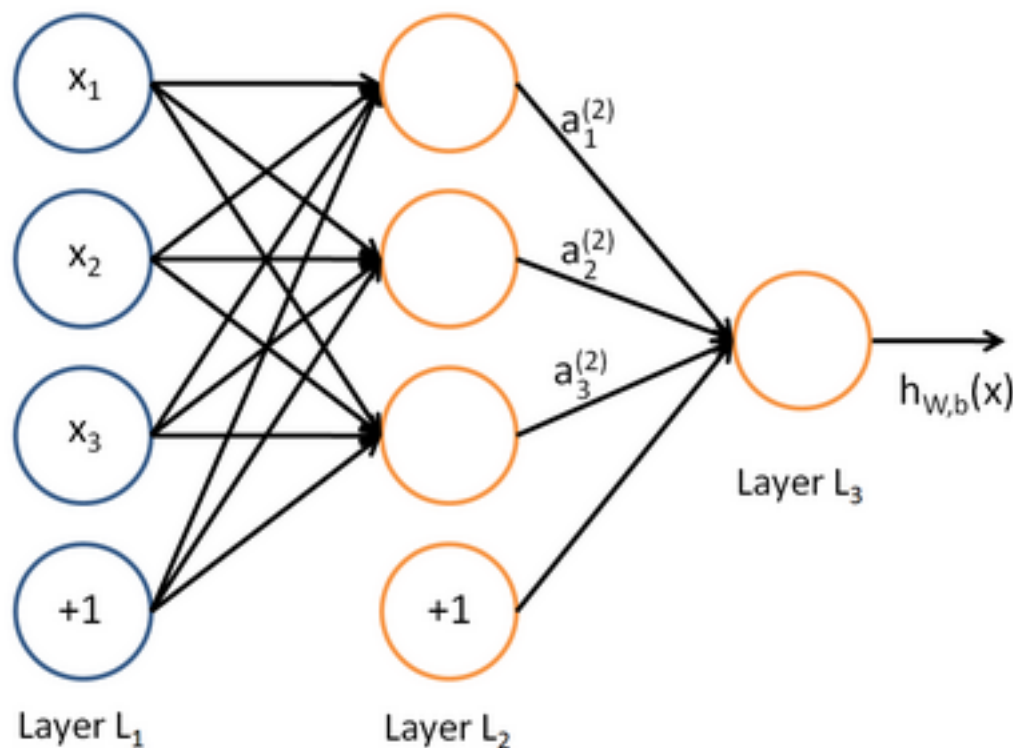


But we don't have to decide ahead of time what variables these logistic regressions are trying to predict!

A neural network

= running several logistic regressions at the same time

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

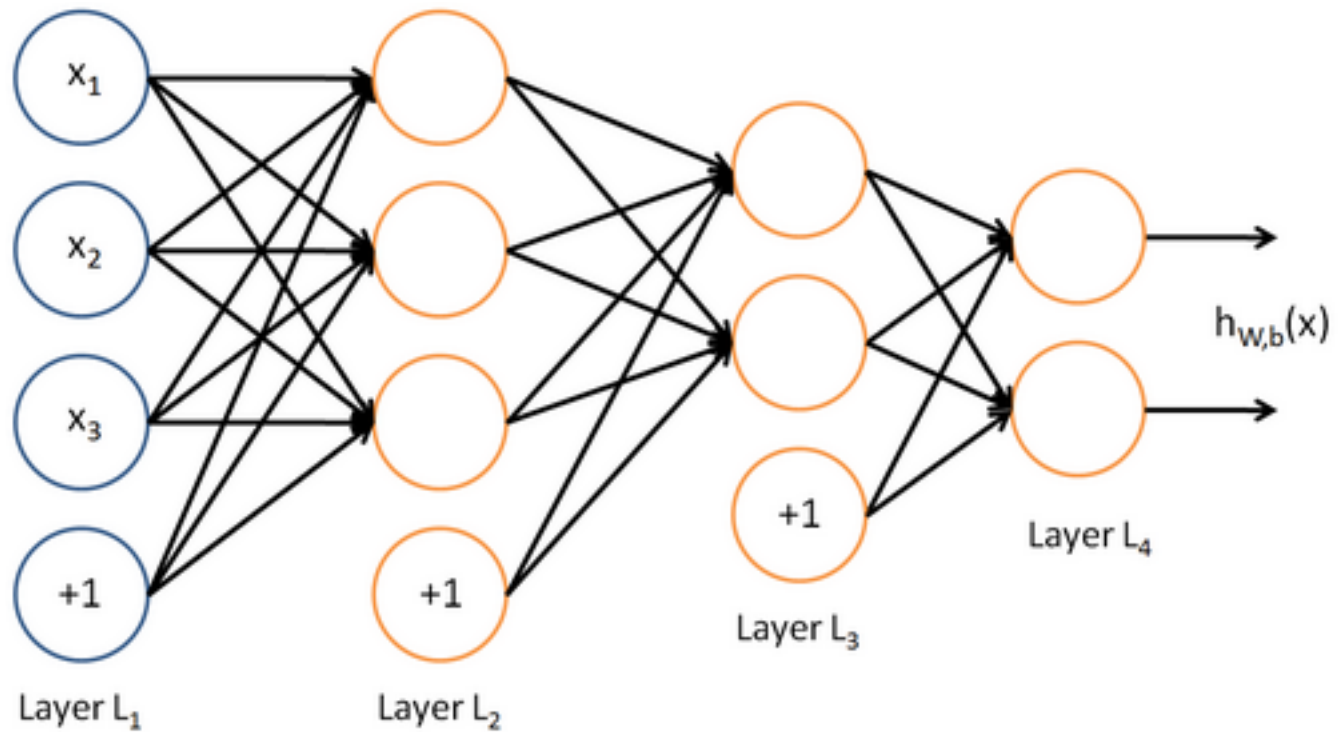


It is the loss function that will direct what the intermediate hidden variables should be, so as to do a good job at predicting the targets for the next layer, etc.

A neural network

= running several logistic regressions at the same time

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



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.

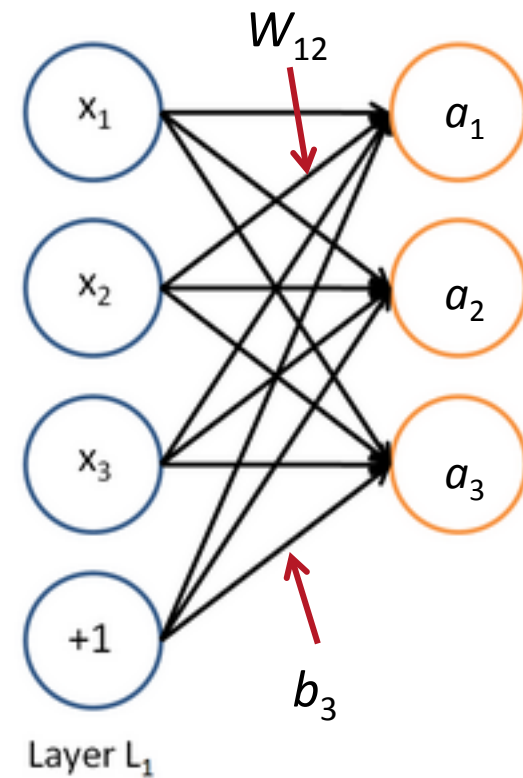
In matrix notation

$$z = Wx + b$$

$$a = f(z)$$

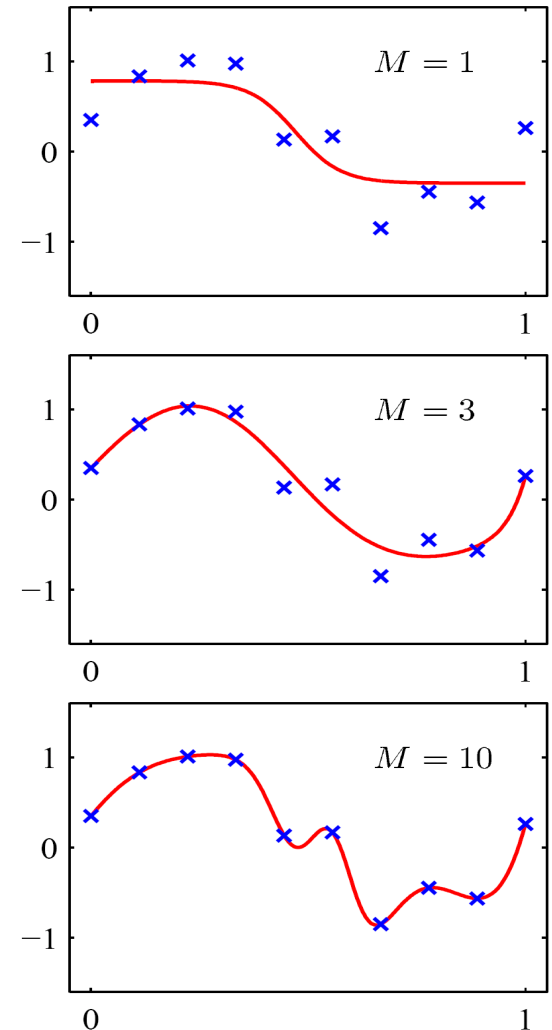
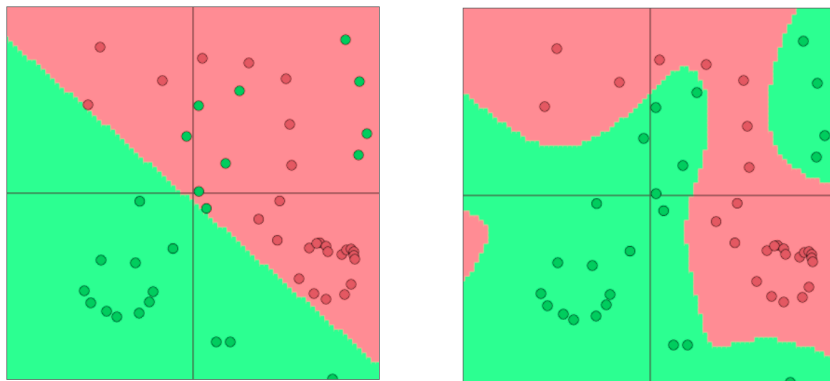
where f is applied element-wise:

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



Non-linearities (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!



A more powerful window classifier

- Revisiting

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

A Single Layer Neural Network

- A single layer is a combination of a linear layer and a nonlinearity:

$$z = Wx + b$$

$$a = f(z)$$

- The neural activations a can then be used to compute some function
- For instance, a softmax probability or an unnormalized score or a we care about:

$$\text{score}(x) = U^T a \in \mathbb{R}$$

Summary: Feed-forward Computation

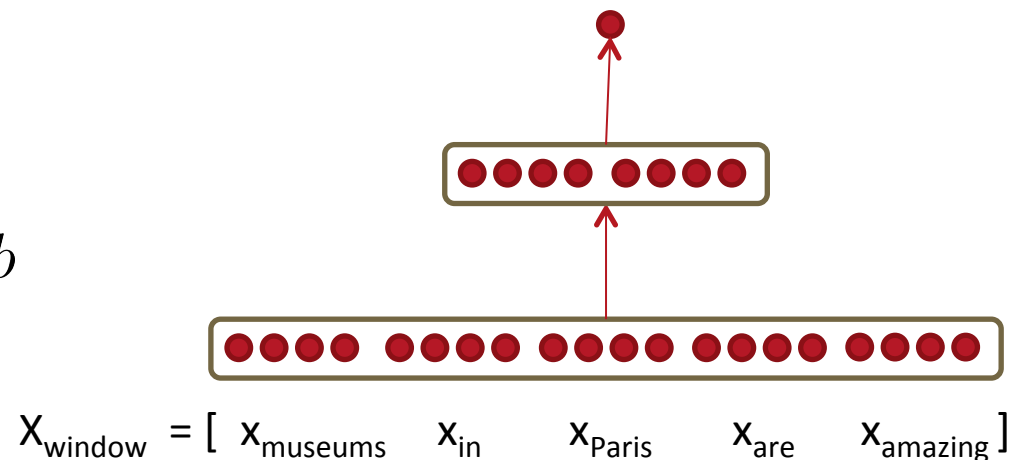
Computing a window's score with a 3-layer neural net: $s = \text{score}(\text{museums in Paris are amazing})$

$$s = U^T f(Wx + b) \quad x \in \mathbb{R}^{20 \times 1}, W \in \mathbb{R}^{8 \times 20}, U \in \mathbb{R}^{8 \times 1}$$

$$s = U^T a$$

$$a = f(z)$$

$$z = Wx + b$$



Next lecture:

Training a window-based neural network.

Taking more **deeper derivatives** → **Backprop**

Then we have all the basic tools in place to learn about more complex models :)

Overview Today:

- Project Ideas
- From one to multi layer neural network!
- Max-Margin loss and **backprop**

Class Project

- Most important (40%) and lasting result of the class
- PSet 3 a little easier to have more time
- Start early and clearly define your task and dataset

- Project types:
 1. Apply existing neural network model to a new task
 2. Come up with a new neural network model

Class Project: Apply Existing NNets to Tasks

1. Define Task:

- Example: **Summarization**

2. Define Dataset

1. Search for academic datasets

- They already have baselines
- E.g.: Document Understanding Conference (DUC)

2. Define your own (harder, need more new baselines)

- If you're a graduate student: connect to your research
- Summarization, Wikipedia: Intro paragraph and rest of large article
- Be creative: Twitter, Blogs, News

Class Project: Apply Existing NNets to Tasks

3. Define your metric

- Search online for well established metrics on this task
- Summarization: Rouge (Recall-Oriented Understudy for Gisting Evaluation) which defines n-gram overlap to human summaries

4. Split your dataset!

- Train/Dev/Test
- Academic dataset often come pre-split
- Don't look at the test split until ~1 week before deadline!

Class Project: Apply Existing NNets to Tasks

5. Establish a baseline

- Implement the simplest model (often logistic regression on unigrams and bigrams) first
- Compute metrics on train AND dev
- Analyze errors
- If metrics are amazing and no errors: done, problem was too easy, restart :)

6. Implement existing neural net model

- Compute metric on train and dev
- Analyze output and errors
- Minimum bar for this class

Class Project: Apply Existing NNets to Tasks

7. Always be close to your data!

- Visualize the dataset
- Collect summary statistics
- Look at errors
- Analyze how different hyperparameters affect performance

8. Try out different model variants

- Soon you will have more options
 - Word vector averaging model (neural bag of words)
 - Fixed window neural model
 - Recurrent neural network
 - Recursive neural network
 - Convolutional neural network

Class Project: A New Model -- Advanced Option

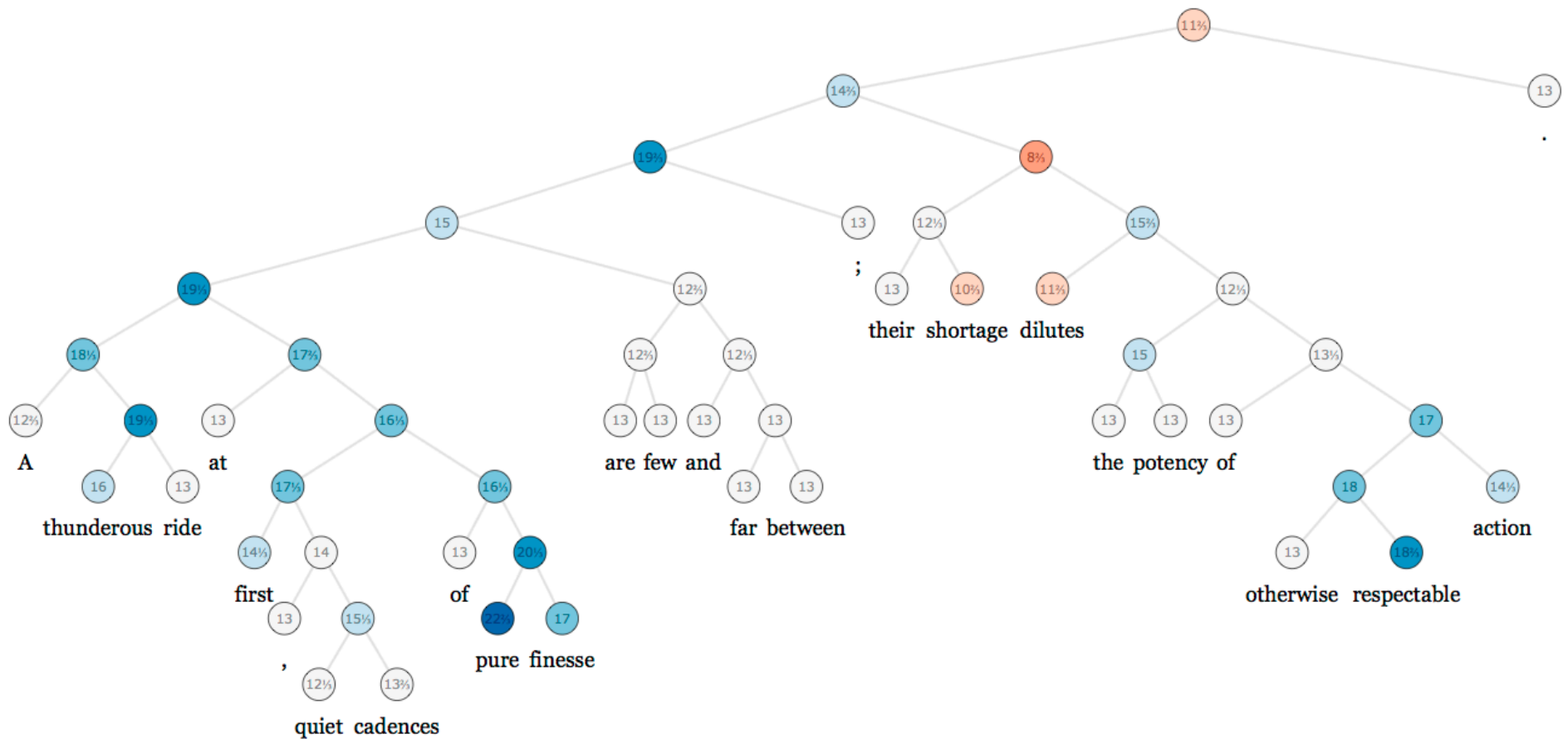
- Do all other steps first (Start early!)
- Gain intuition of why existing models are flawed
- Talk to other researchers, come to my office hours a lot
- Implement new models and iterate quickly over ideas
- Set up efficient experimental framework
- Build simpler new models first
- Example Summarization:
 - Average word vectors per paragraph, then greedy search
 - Implement language model or autoencoder (introduced later)
 - Stretch goal for potential paper: Generate summary!

Project Ideas

- Summarization
- NER, like PSet 2 but with larger data
Natural Language Processing (almost) from Scratch, Ronan Collobert, Jason Weston, Leon Bottou, Michael Karlen, Koray Kavukcuoglu, Pavel Kuksa, <http://arxiv.org/abs/1103.0398>
- Simple question answering,
[A Neural Network for Factoid Question Answering over Paragraphs](#), Mohit Iyyer, Jordan Boyd-Graber, Leonardo Claudino, Richard Socher and Hal Daumé III (**EMNLP 2014**)
- Image to text mapping or generation,
[Grounded Compositional Semantics for Finding and Describing Images with Sentences](#), Richard Socher, Andrej Karpathy, Quoc V. Le, Christopher D. Manning, Andrew Y. Ng. (**TACL 2014**)
or
Deep Visual-Semantic Alignments for Generating Image Descriptions, Andrej Karpathy, Li Fei-Fei
- Entity level sentiment
- Use DL to solve an NLP challenge on kaggle,
Develop a scoring algorithm for student-written short-answer responses, <https://www.kaggle.com/c/asap-sas>

Default project: sentiment classification

- Sentiment on movie reviews: <http://nlp.stanford.edu/sentiment/>
- Lots of deep learning baselines and methods have been tried



A more powerful window classifier

- Revisiting
- $X_{\text{window}} = [x_{\text{museums}} \quad x_{\text{in}} \quad x_{\text{Paris}} \quad x_{\text{are}} \quad x_{\text{amazing}}]$
- Assume we want to classify whether the center word is a location or not

A Single Layer Neural Network

- A single layer is a combination of a linear layer and a nonlinearity:

$$z = Wx + b$$

$$a = f(z)$$

- The neural activations a can then be used to compute some function
- For instance, an unnormalized score or a softmax probability we care about:

$$\text{score}(x) = U^T a \in \mathbb{R}$$

Summary: Feed-forward Computation

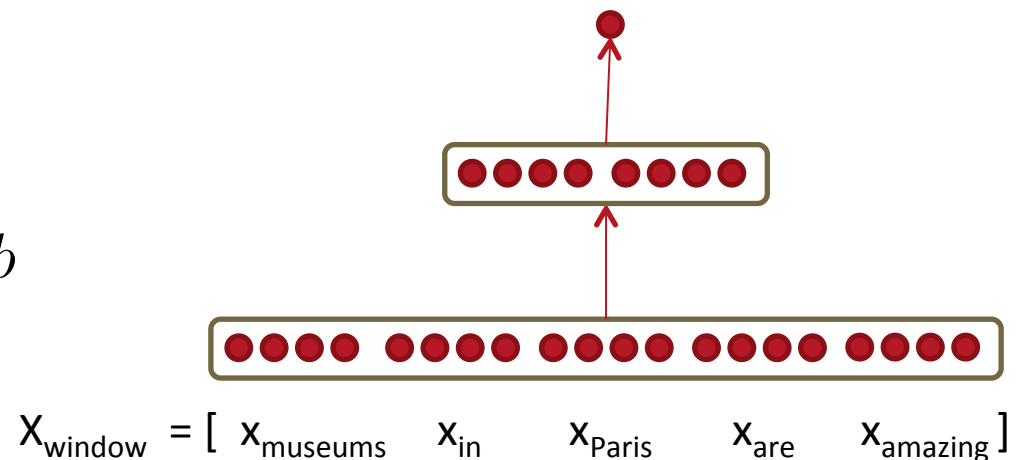
Computing a window's score with a 3-layer neural net: $s = \text{score}(\text{museums in Paris are amazing})$

$$s = U^T f(Wx + b) \quad x \in \mathbb{R}^{20 \times 1}, W \in \mathbb{R}^{8 \times 20}, U \in \mathbb{R}^{8 \times 1}$$

$$s = U^T a$$

$$a = f(z)$$

$$z = Wx + b$$

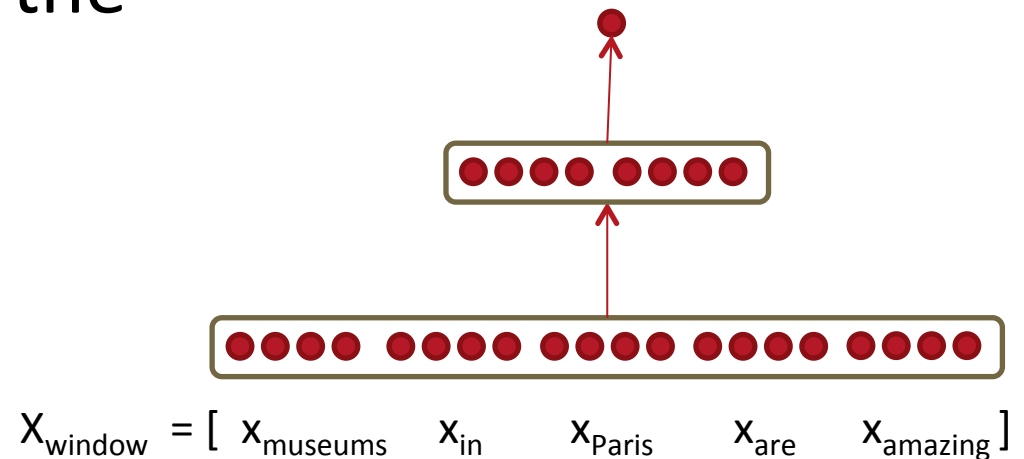


Main intuition for extra layer

The layer learns non-linear interactions between the input word vectors.

Example:

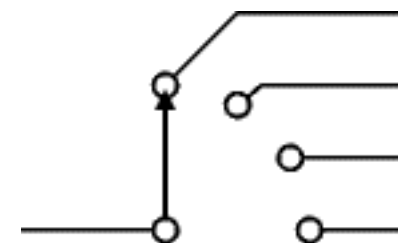
only if “*museums*” is first vector should it matter that “*in*” is in the second position



Summary: Feed-forward Computation

- $s = \text{score}(\text{museums in Paris are amazing})$
- $s_c = \text{score}(\text{Not all museums in Paris})$
- Idea for training objective: make score of true window larger and corrupt window's score lower (until they're good enough): minimize

$$J = \max(0, 1 - s + s_c)$$



- This is continuous, can perform SGD

Max-margin Objective function

- Objective for a single window:

$$J = \max(0, 1 - s + s_c)$$

- Each window with a location at its center should have a score +1 higher than any window without a location at its center
- xxx |← 1 →| ooo
- For full objective function: Sum over all training windows

Training with Backpropagation

$$J = \max(0, 1 - s + s_c)$$

$$s = U^T f(Wx + b)$$

$$s_c = U^T f(Wx_c + b)$$

Assuming cost J is > 0 ,

compute the derivatives of s and s_c wrt all the involved variables: U, W, b, x

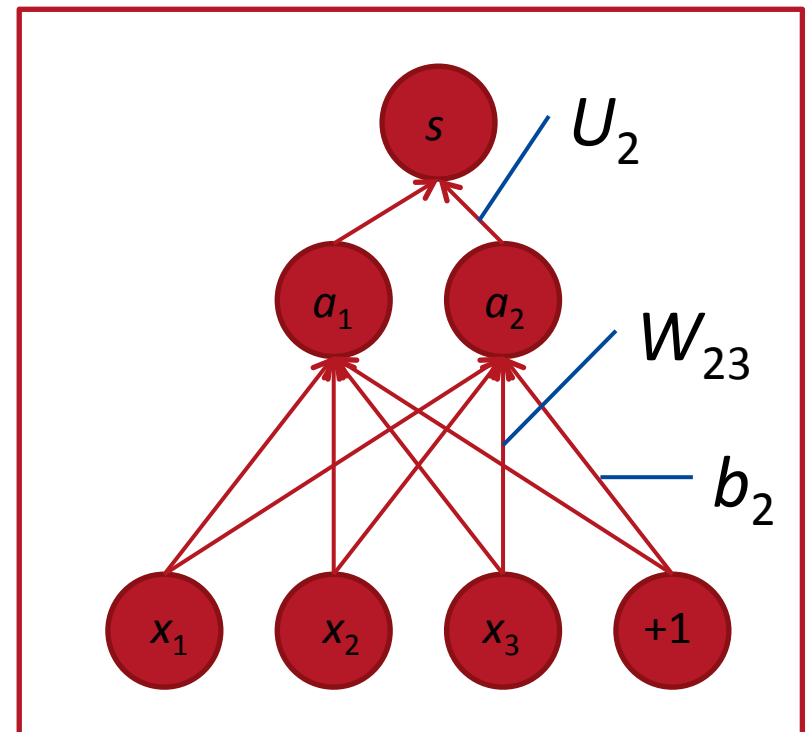
$$\frac{\partial s}{\partial U} = \frac{\partial}{\partial U} U^T a \qquad \frac{\partial s}{\partial U} = a$$

Training with Backpropagation

- Let's consider the derivative of a single weight W_{ij}

$$\frac{\partial s}{\partial W} = \frac{\partial}{\partial W} U^T a = \frac{\partial}{\partial W} U^T f(z) = \frac{\partial}{\partial W} U^T f(Wx + b)$$

- This only appears inside a_i
- For example: W_{23} is only used to compute a_2



Training with Backpropagation

$$\frac{\partial s}{\partial W} = \frac{\partial}{\partial W} U^T a = \frac{\partial}{\partial W} U^T f(z) = \frac{\partial}{\partial W} U^T f(Wx + b)$$

Derivative of weight W_{ij} :

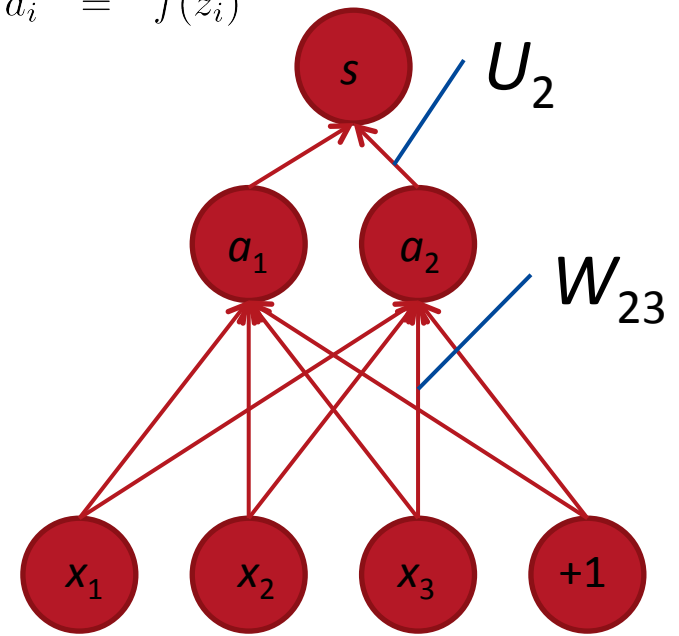
$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \frac{\partial u}{\partial x}$$

$$\frac{\partial}{\partial W_{ij}} U^T a \rightarrow \frac{\partial}{\partial W_{ij}} U_i a_i$$

$$z_i = W_i \cdot x + b_i = \sum_{j=1}^3 W_{ij} x_j + b_i$$

$$a_i = f(z_i)$$

$$\begin{aligned} U_i \frac{\partial}{\partial W_{ij}} a_i &= U_i \frac{\partial a_i}{\partial z_i} \frac{\partial z_i}{\partial W_{ij}} \\ &= U_i \frac{\partial f(z_i)}{\partial z_i} \frac{\partial z_i}{\partial W_{ij}} \\ &= U_i f'(z_i) \frac{\partial z_i}{\partial W_{ij}} \\ &= U_i f'(z_i) \frac{\partial W_i \cdot x + b_i}{\partial W_{ij}} \end{aligned}$$



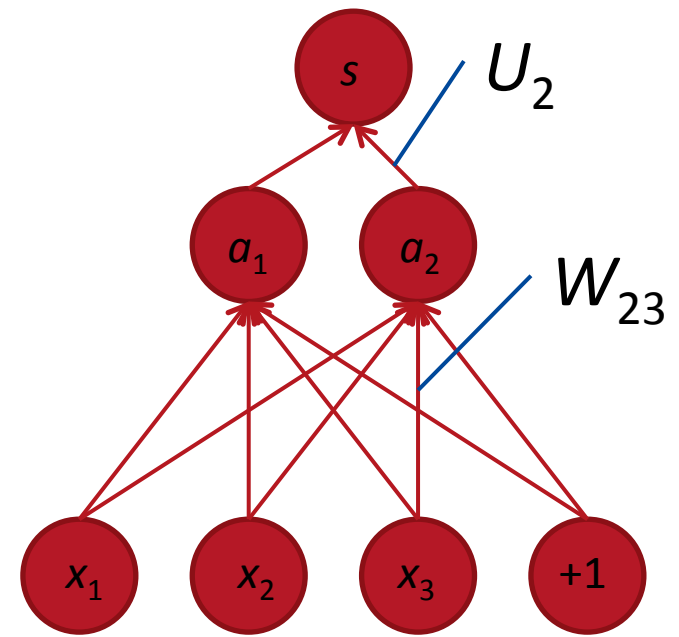
Training with Backpropagation

Derivative of single weight W_{ij} :

$$\begin{aligned}
 U_i \frac{\partial}{\partial W_{ij}} a_i &= U_i f'(z_i) \frac{\partial W_i \cdot x + b_i}{\partial W_{ij}} \\
 &= U_i f'(z_i) \frac{\partial}{\partial W_{ij}} \sum_k W_{ik} x_k \\
 &= \underbrace{U_i f'(z_i)}_{\delta_i} x_j \\
 &= \underbrace{\delta_i}_{\text{Local error signal}} \underbrace{x_j}_{\text{Local input signal}}
 \end{aligned}$$

where $f'(z) = f(z)(1 - f(z))$ for logistic f

$$\begin{aligned}
 z_i &= W_i \cdot x + b_i = \sum_{j=1}^3 W_{ij} x_j + b_i \\
 a_i &= f(z_i)
 \end{aligned}$$



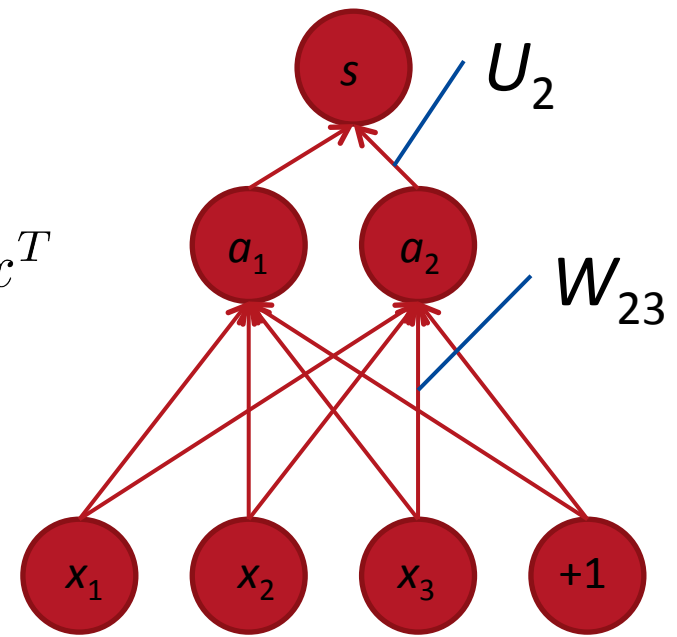
Training with Backpropagation

- From single weight W_{ij} to full W :

$$\begin{aligned} \frac{\partial s}{\partial W_{ij}} &= \underbrace{U_i f'(z_i)}_{\delta_i} x_j \\ &= \delta_i x_j \end{aligned}$$

$$\begin{aligned} z_i &= W_i \cdot x + b_i = \sum_{j=1}^3 W_{ij} x_j + b_i \\ a_i &= f(z_i) \end{aligned}$$

- We want all combinations of $i = 1, 2$ and $j = 1, 2, 3 \rightarrow ?$
- Solution: Outer product: $\frac{\partial J}{\partial W} = \delta x^T$ where $\delta \in \mathbb{R}^{2 \times 1}$ is the “responsibility” or error message coming from each activation a



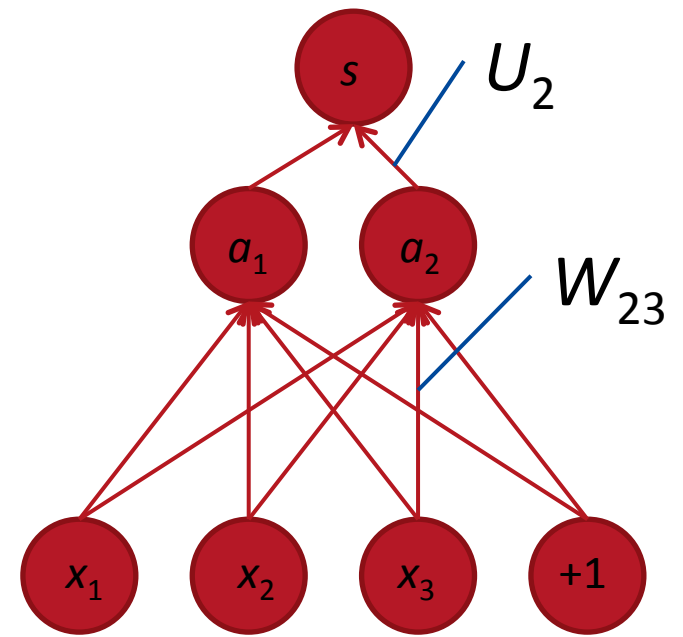
Training with Backpropagation

- For biases b , we get:

$$z_i = W_i \cdot x + b_i = \sum_{j=1}^3 W_{ij} x_j + b_i$$

$$a_i = f(z_i)$$

$$\begin{aligned} & U_i \frac{\partial}{\partial b_i} a_i \\ = & U_i f'(z_i) \frac{\partial W_i \cdot x + b_i}{\partial b_i} \\ = & \delta_i \end{aligned}$$



Training with Backpropagation

That's almost backpropagation

It's simply taking derivatives and using the chain rule!

Remaining trick: we can **re-use** derivatives computed for higher layers in computing derivatives for lower layers!

Example: last derivatives of model, the word vectors in x

Training with Backpropagation

- Take derivative of score with respect to single element of word vector
- Now, we cannot just take into consideration one a_i because each x_j is connected to all the neurons above and hence x_j influences the overall score through all of these, hence:

$$\begin{aligned}
 \frac{\partial s}{\partial x_j} &= \sum_{i=1}^2 \frac{\partial s}{\partial a_i} \frac{\partial a_i}{\partial x_j} \\
 &= \sum_{i=1}^2 \frac{\partial U^T a}{\partial a_i} \frac{\partial a_i}{\partial x_j} \\
 &= \sum_{i=1}^2 U_i \frac{\partial f(W_i \cdot x + b)}{\partial x_j} \\
 &= \sum_{i=1}^2 \underbrace{U_i f'(W_i \cdot x + b)}_{\delta_i} \frac{\partial W_i \cdot x}{\partial x_j} \\
 &= \sum_{i=1}^2 \delta_i W_{ij} \\
 &= W_{\cdot j}^T \delta
 \end{aligned}$$

Re-used part of previous derivative 

Training with Backpropagation

- With $\frac{\partial s}{\partial x_j} = W_{\cdot j}^T \delta$, what is the full gradient? \rightarrow

$$\frac{\partial s}{\partial x} = W^T \delta$$

- Observations: The error message δ that arrives at a hidden layer has the same dimensionality as that hidden layer

Putting all gradients together:

- Remember: Full objective function for each window was:

$$J = \max(0, 1 - s + s_c)$$

$$s = U^T f(Wx + b)$$
$$s_c = U^T f(Wx_c + b)$$

- For example: gradient for U:

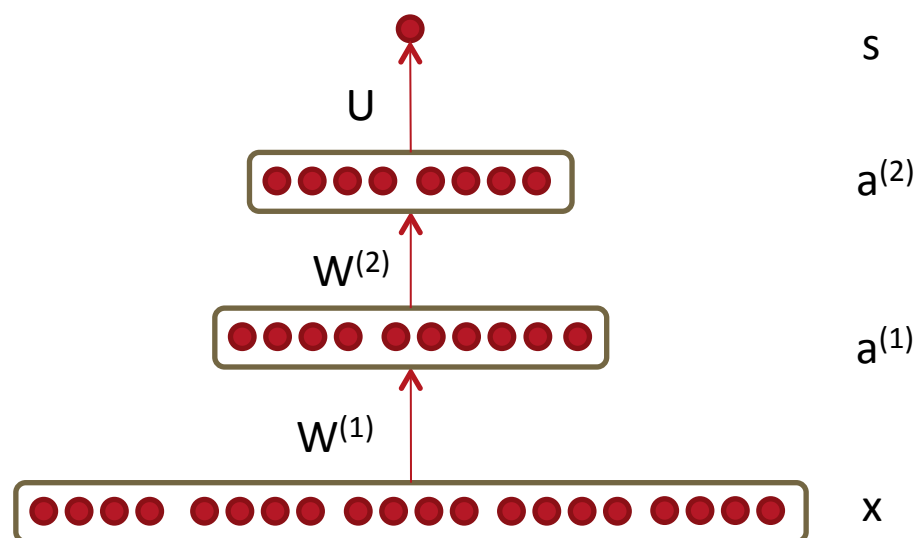
$$\frac{\partial s}{\partial U} = 1\{1 - s + s_c > 0\} (-f(Wx + b) + f(Wx_c + b))$$

$$\frac{\partial s}{\partial U} = 1\{1 - s + s_c > 0\} (-a + a_c)$$

Two layer neural nets and full backprop

- Let's look at a 2 layer neural network
- Same window definition for x
- Same scoring function
- 2 hidden layers

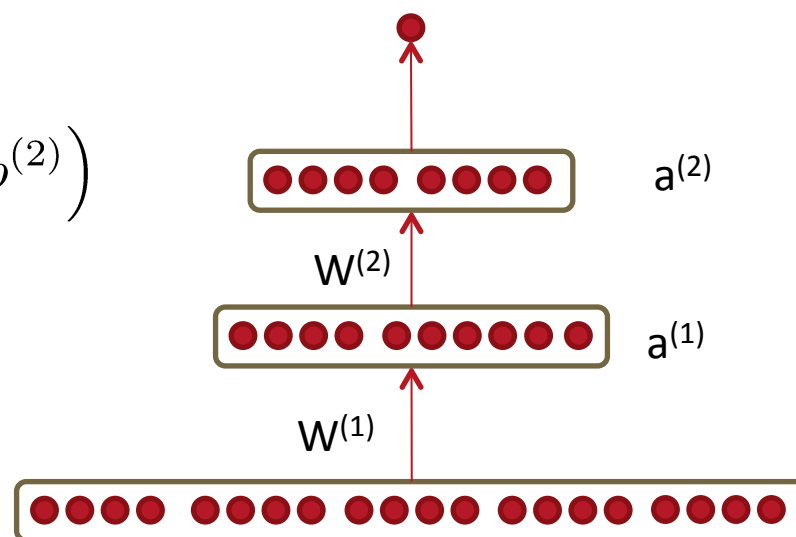
$$\begin{aligned}z^{(1)} &= W^{(1)}x + b^{(1)} \\ a^{(1)} &= f\left(z^{(1)}\right) \\ z^{(2)} &= W^{(2)}a^{(1)} + b^{(2)} \\ a^{(2)} &= f\left(z^{(2)}\right) \\ s &= U^T a^{(2)}\end{aligned}$$



Two layer neural nets and full backprop

- Fully written out as one function:

$$\begin{aligned}
 s &= U^T f \left(W^{(2)} f \left(W^{(1)} x + b^{(1)} \right) + b^{(2)} \right) \\
 &= U^T a^{(2)} \\
 &= U^T f \left(W^{(2)} a^{(1)} + b^{(2)} \right)
 \end{aligned}$$



- Same derivation as before for $W^{(2)}$ (now sitting on $a^{(1)}$)

$$\begin{aligned}
 \frac{\partial s}{\partial W_{ij}^{(2)}} &= \underbrace{U_i f'(z_i)}_{\delta_i} x_j & \frac{\partial s}{\partial W_{ij}^{(2)}} &= \underbrace{U_i f'(z_i^{(2)})}_{\delta_i^{(2)}} a_j^{(1)} \\
 &= \delta_i x_j & &= \delta_i^{(2)} a_j^{(1)}
 \end{aligned}$$

Two layer neural nets and full backprop

- Same derivation as before for $W^{(2)}$:

$$\begin{aligned}\frac{\partial s}{\partial W_{ij}^{(2)}} &= \underbrace{U_i f' \left(z_i^{(2)} \right)}_{\delta_i^{(2)}} a_j^{(1)} \\ &= \delta_i^{(2)} a_j^{(1)}\end{aligned}$$

$$\begin{aligned}z^{(1)} &= W^{(1)}x + b^{(1)} \\ a^{(1)} &= f \left(z^{(1)} \right) \\ z^{(2)} &= W^{(2)}a^{(1)} + b^{(2)} \\ a^{(2)} &= f \left(z^{(2)} \right) \\ s &= U^T a^{(2)}\end{aligned}$$

- In matrix notation: $\frac{\partial s}{\partial W^{(2)}} = \delta^{(2)} a^{(1)T}$

where $\delta^{(2)} = U \circ f' \left(z^{(2)} \right)$ and \circ is the element-wise product also called Hadamard product

- Last missing piece for understanding general backprop: $\frac{\partial s}{\partial W^{(1)}}$

Two layer neural nets and full backprop

- Last missing piece: $\frac{\partial s}{\partial W^{(1)}}$
 - What's the last layer's error message $\delta^{(1)}$?
- $$\begin{aligned}z^{(1)} &= W^{(1)}x + b^{(1)} \\a^{(1)} &= f\left(z^{(1)}\right) \\z^{(2)} &= W^{(2)}a^{(1)} + b^{(2)} \\a^{(2)} &= f\left(z^{(2)}\right) \\s &= U^T a^{(2)}\end{aligned}$$

- Similar derivation to single layer model
- Main difference, we have both $W_{\cdot j}^{(1)T} \delta^{(2)}$ and $f'(z^{(1)})$ because unlike x , $W^{(1)}$ is inside another function \rightarrow chain rule

Two layer neural nets and full backprop

- Difference for δ : we have both $W_{.j}^{(1)T} \delta^{(2)}$ and $f'(z^{(1)})$ because unlike x , $W^{(1)}$ is inside another function \rightarrow chain rule

$$\begin{aligned} \dots &= \left((\delta^{(n_l)})^T W_{.i}^{(n_l-1)} \right) f'(z_i^{(n_l-1)}) a_j^{(n_l-2)} \\ &= \underbrace{\left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(n_l-1)} \delta_j^{(n_l)} \right) f'(z_i^{(n_l-1)}) a_j^{(n_l-2)}} \\ &= \delta_i^{(n_l-1)} a_j^{(n_l-2)} \end{aligned}$$

- Putting it all together: $\delta^{(1)} = \left(W^{(1)T} \delta^{(2)} \right) \circ f' \left(z^{(1)} \right)$

Two layer neural nets and full backprop

- Last missing piece: $\frac{\partial s}{\partial W^{(1)}} = \delta^{(1)} x^T$
- In general for any matrix $W^{(l)}$ at internal layer l and any error with regularization E_R all backprop in standard multilayer neural networks boils down to 2 equations:

$$\begin{aligned}z^{(1)} &= W^{(1)}x + b^{(1)} \\a^{(1)} &= f\left(z^{(1)}\right) \\z^{(2)} &= W^{(2)}a^{(1)} + b^{(2)} \\a^{(2)} &= f\left(z^{(2)}\right) \\s &= U^T a^{(2)}\end{aligned}$$

$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)} \right) \circ f'(z^{(l)}),$$

$$\frac{\partial}{\partial W^{(l)}} E_R = \delta^{(l+1)} (a^{(l)})^T + \lambda W^{(l)}$$

- Top and bottom layers have simpler δ

Backpropagation (High Level)

Back-Prop

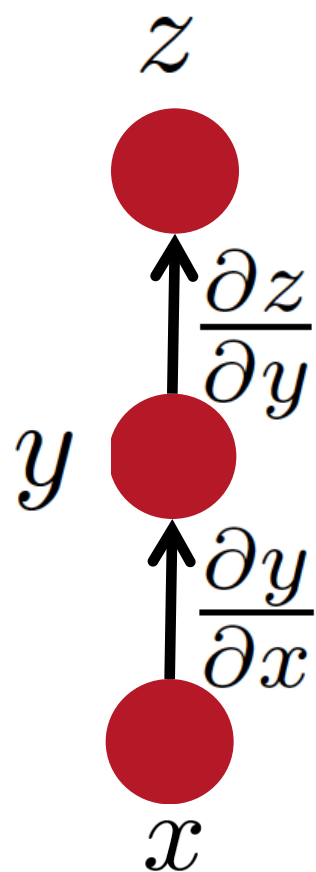
- Compute gradient of example-wise loss wrt parameters

- Simply applying the derivative chain rule wisely

$$z = f(y) \quad y = g(x) \quad \frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$$

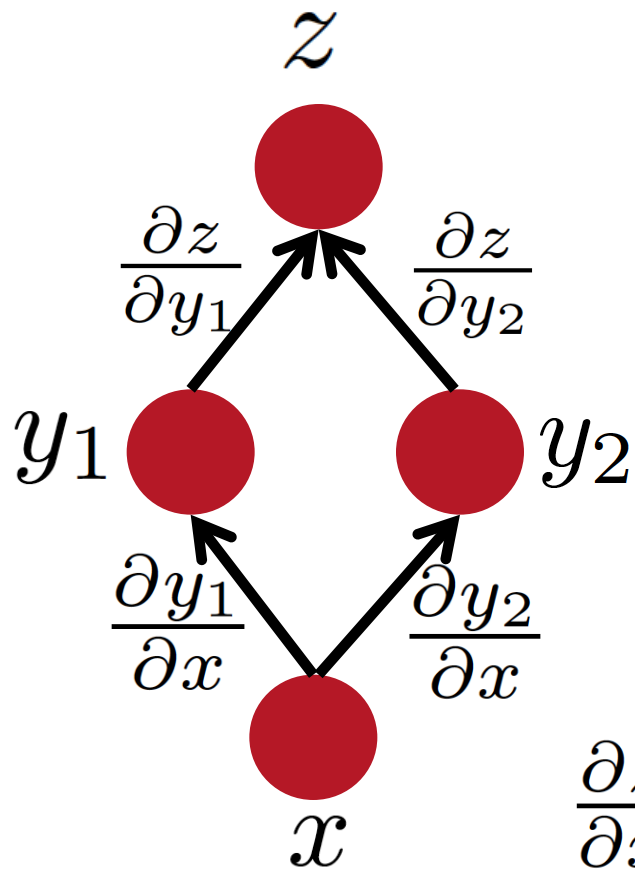
- If computing the loss(example, parameters) is $O(n)$ computation, then so is computing the gradient

Simple Chain Rule



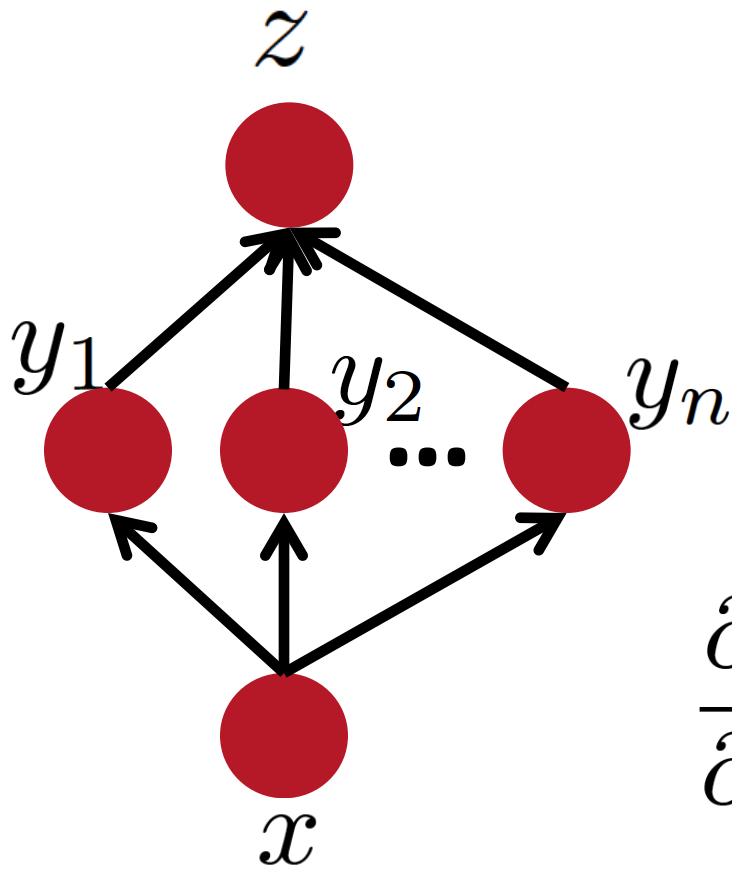
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$$

Multiple Paths Chain Rule



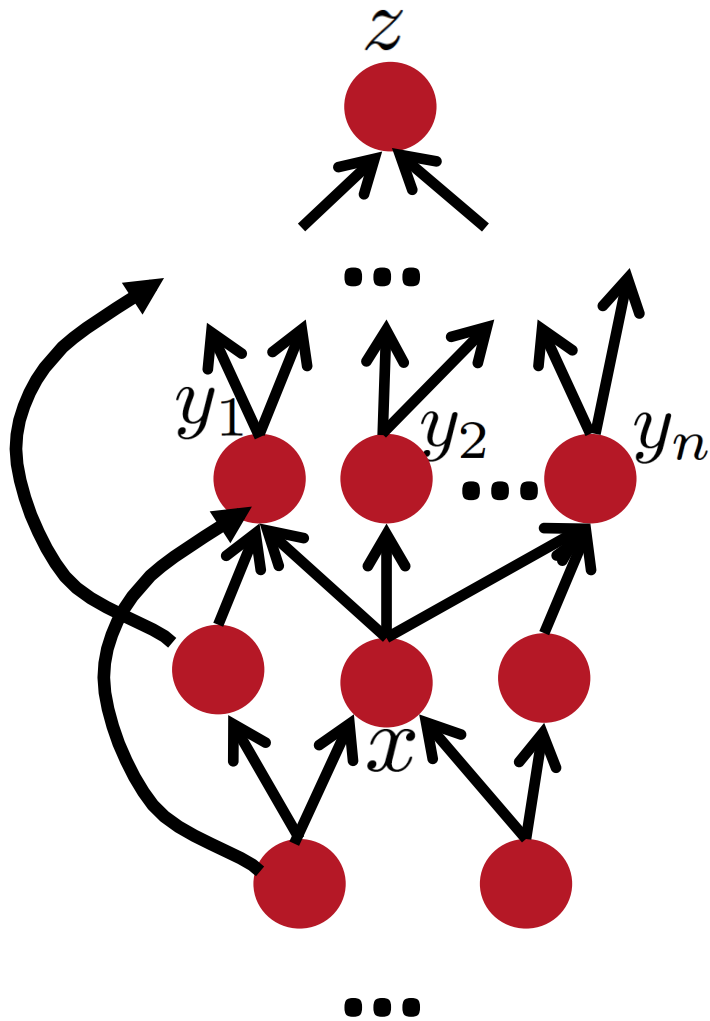
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1} \frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2} \frac{\partial y_2}{\partial x}$$

Multiple Paths Chain Rule - General



$$\frac{\partial z}{\partial x} = \sum_{i=1}^n \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

Chain Rule in Flow Graph

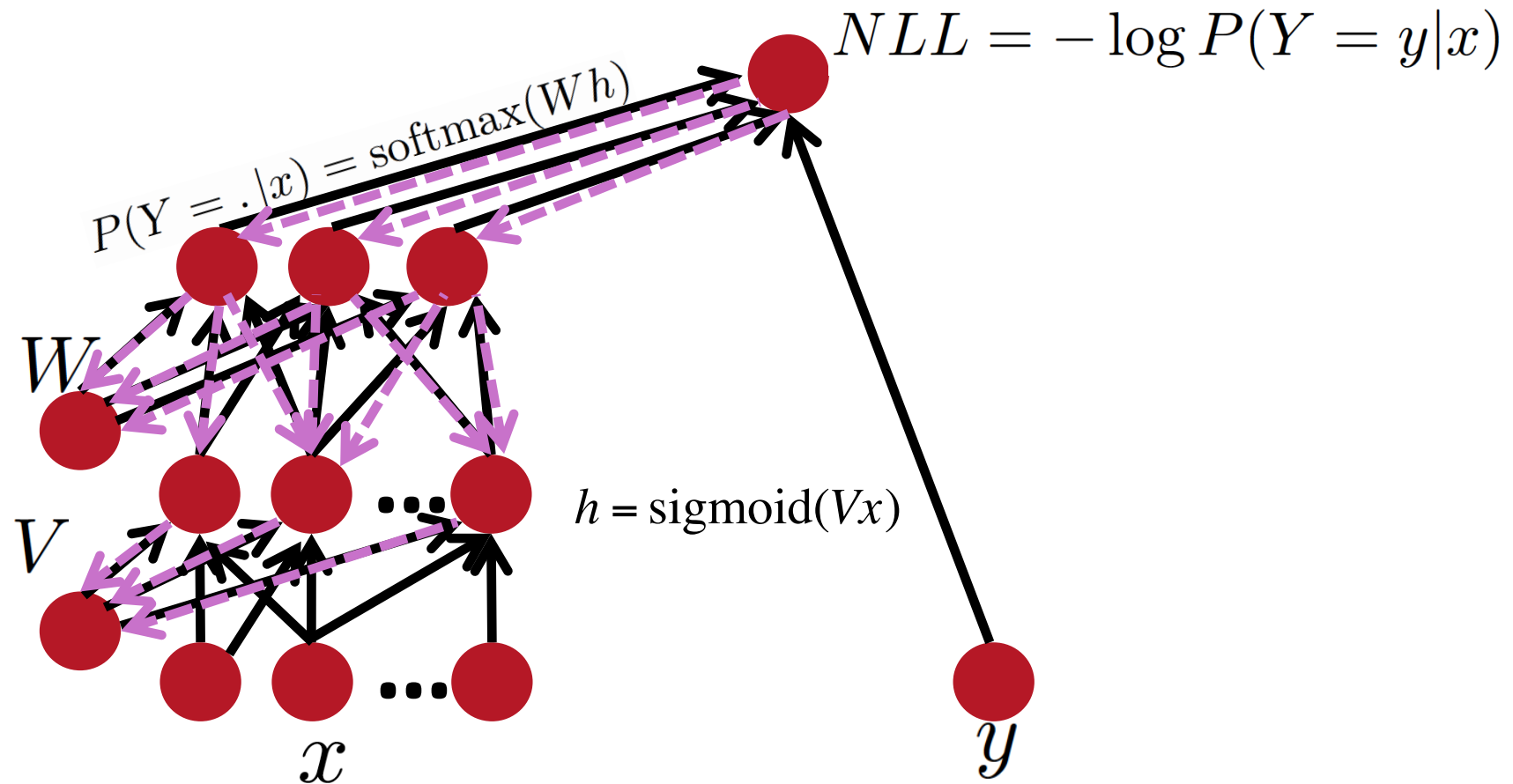


Flow graph: any directed acyclic graph
node = computation result
arc = computation dependency

$\{y_1, y_2, \dots, y_n\}$ = successors of x

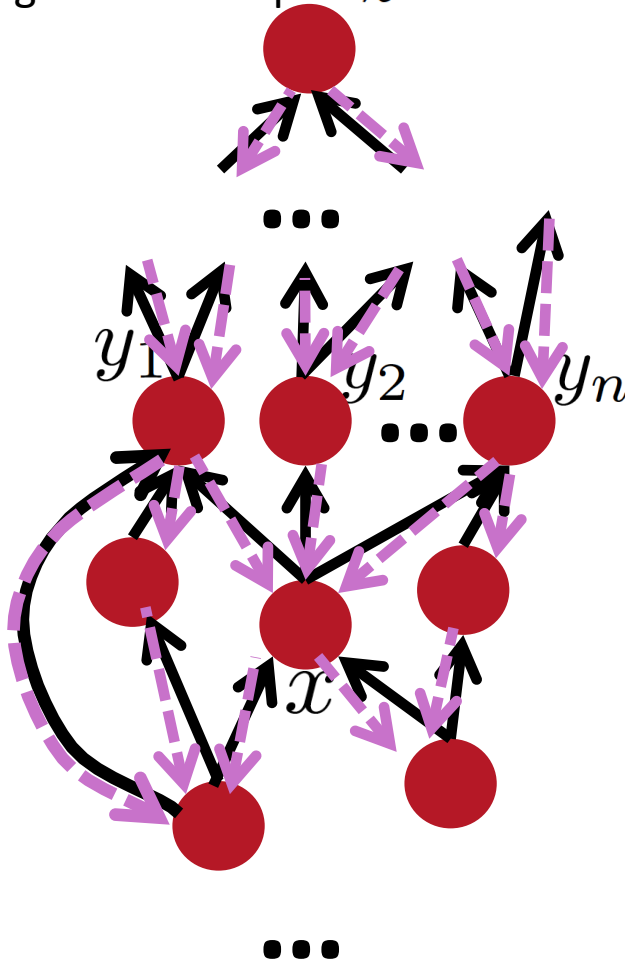
$$\frac{\partial z}{\partial x} = \sum_{i=1}^n \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

Back-Prop in Multi-Layer Net



Back-Prop in General Flow Graph

Single scalar output z

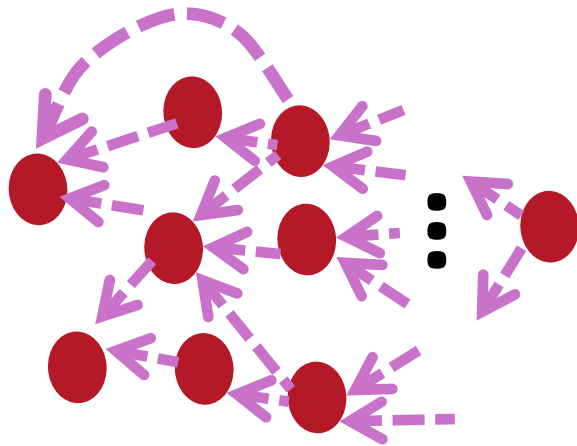
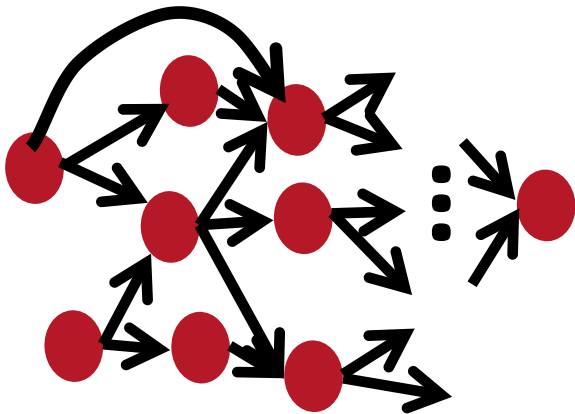


1. Fprop: visit nodes in topo-sort order
 - Compute value of node given predecessors
2. Bprop:
 - initialize output gradient = 1
 - visit nodes in reverse order:
 - Compute gradient wrt each node using gradient wrt successors

$\{y_1, y_2, \dots, y_n\}$ = successors of x

$$\frac{\partial z}{\partial x} = \sum_{i=1}^n \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

Automatic Differentiation



- The gradient computation can be **automatically inferred** from the symbolic expression of the fprop.
- Each node type needs to know how to compute its output and how to compute the gradient wrt its inputs given the gradient wrt its output.
- Easy and fast prototyping

Summary

- Congrats!
- You survived the hardest part of this class.
- Everything else from now on is just more matrix multiplications and backprop :)

- Next up:
 - Tips and Tricks
 - Recurrent Neural Networks