Natural Language Processing with Deep Learning

CS224N/Ling284

Christopher Manning

Lecture 1: Introduction and Word Vectors

Lecture Plan

Lecture 1: Introduction and Word Vectors

1. The course (10 mins)
2. Human language and word meaning (15 mins)
3. Word2vec introduction (15 mins)
4. Word2vec objective function gradients (25 mins)
5. Optimization basics (5 mins)
6. Looking at word vectors (10 mins or less)

Course logistics in brief

- Instructor: Christopher Manning
- Head TA and co-instructor: Abigail See
- TAs: Many wonderful people! See website
- Time: TuTh 4:30–5:50, Nvidia Aud (→ video)

- Other information: see the class webpage:
  - http://cs224n.stanford.edu/
    a.k.a., http://www.stanford.edu/class/cs224n/
  - Syllabus, **office hours**, "handouts", TAs, Piazza
    - Office hours start **this Thursday**
  - Slides uploaded before each lecture

What do we hope to teach?

1. An understanding of the effective modern methods for deep learning
   - Basics first, then key methods used in NLP: Recurrent networks, attention, etc. (①, ②; widely used for NLP models)
2. A big picture understanding of human languages and the difficulties in understanding and producing them (overview)
3. An understanding of and ability to build systems (in PyTorch) for some of the major problems in NLP: (Practical part)
   - Word meaning, dependency parsing, machine translation, question answering

What's different this year?

new content, area of deep learning evolving quickly.

- Lectures (including guest lectures) covering new material: character models, transformers, safety/fairness, multitask learn.
- 5x one-week assignments instead of 3x two-week assignments
- Assignments covering new material (NMT with attention, ConvNets, subword modeling)
- Using PyTorch rather than TensorFlow
- Assignments due *before class* (4:30pm) not at midnight!
- Gentler but earlier ramp-up
  - First assignment is easy, but due *one week from today*!
- No midterm

Course work and grading policy

- 5 x 1-week Assignments: 6% + 4 x 12%: 54%
  - HW1 is released today! Due next Tuesday! At 4:30 p.m.
  - Please use @stanford.edu email for your Gradescope account
- Final Default or Custom Course Project (1–3 people): 43%
  - Project proposal: 5%, milestone: 5%, poster: 3%, report: 30%
  - Final poster session attendance expected! (See website.)
  - **Wed Mar 20, 5pm-10pm** (put it in your calendar!)
- Participation: 3%
  - (Guest) lecture attendance, Piazza, eval, karma – see website!
- Late day policy
  - 6 free late days; afterwards, 10% off per day late
  - Assignments not accepted after 3 late days per assignment
- Collaboration policy: Read the website and the Honor Code! Understand allowed 'collaboration' and how to document it

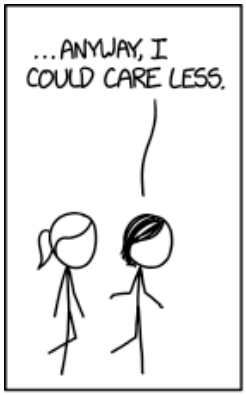
High-Level Plan for Problem Sets

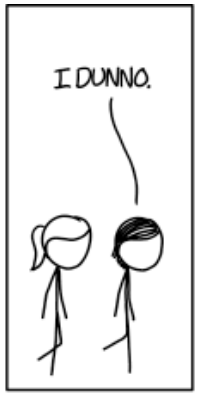
- HW1 is hopefully an easy on ramp – an IPython Notebook
- HW2 is pure Python (numpy) but expects you to do (multivariate) calculus so you really understand the basics
- HW3 introduces PyTorch
- HW4 and HW5 use PyTorch on a GPU (Microsoft Azure)
  - Libraries like PyTorch, Tensorflow (and Chainer, MXNet, CNTK, Keras, etc.) are becoming the standard tools of DL
- For FP, you either
  - Do the default project, which is SQuAD question answering
    - Open-ended but an easier start; a good choice for most
  - Propose a custom final project, which we approve
    - You will receive feedback from a **mentor** (TA/prof/postdoc/PhD)
  - Can work in teams of 1–3; can use any language

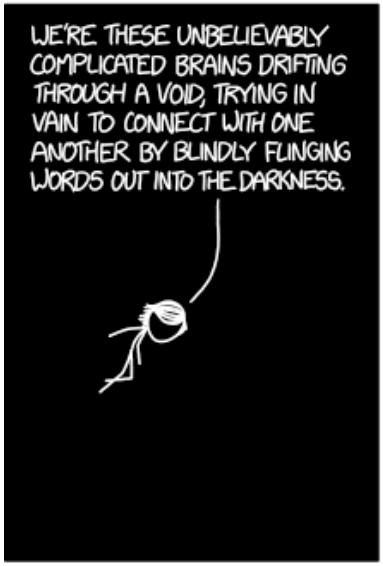
Lecture Plan

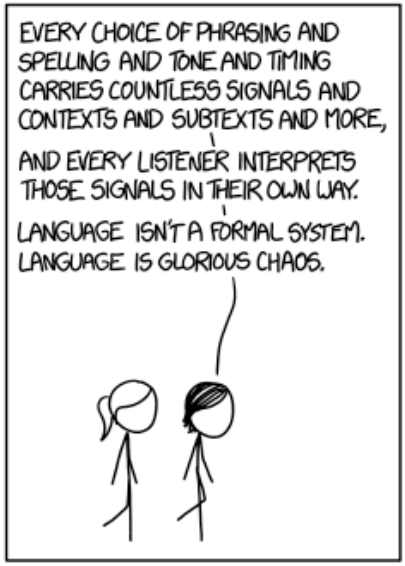
1. The course (10 mins)
2. Human language and word meaning (15 mins)
3. Word2vec introduction (15 mins)
4. Word2vec objective function gradients (25 mins)
5. Optimization basics (5 mins)
6. Looking at word vectors (10 mins or less)

2 anecdotes about human language:





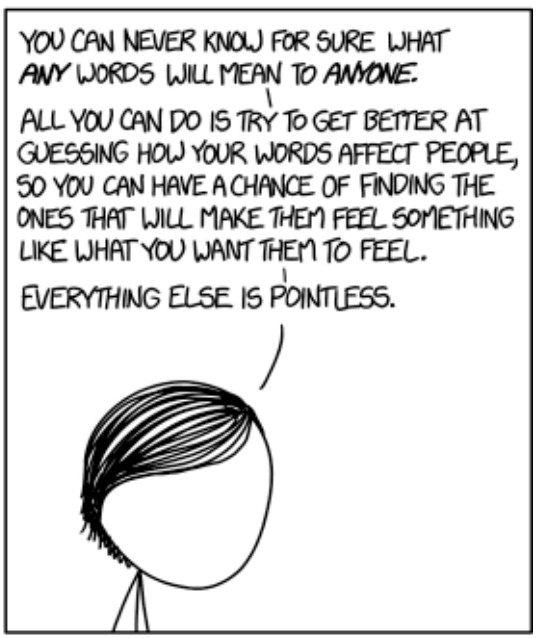


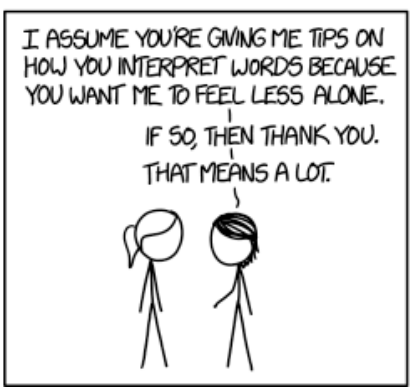


→ how language is uncertain, evolved system of communication, but we have enough agreed meaning we can somehow communicate.

→ but we are some kind of probabilistic inference of what people mean.

→ and we are using language not just for info. functions but for social functions.






AI to get intelligent
→ get comps. have the knowledge of human beings,
→ human beings have knowledge that gives them intelligence.
→ most of knowledge in your head is tacit communicating to you.
→ development of language makes human invincible
→ language communication incredibly slow 15 words/min but you can assume other person knows how to interpret
ANNO
LITERARIA INSTRUXIT

1. How do we represent the meaning of a word?

words have rich meaning.

Definition: **meaning** (Webster dictionary)

relates meaning to idea

- the idea that is represented by a word, phrase, etc.

def^n's kind of a cop out, rewriting meaning in terms of the word idea.

- 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.

Commonest linguistic way of thinking of meaning:

signifier (symbol) ⟺ signified (idea or thing)

→ what thing it represents = denotational semantics

→ hard thing to get your hands on computationally

How do we have usable meaning in a computer?

Working out meaning on a computer:

Common solution: Use e.g. WordNet, a thesaurus containing lists of **synonym sets** and **hypernyms** ("is a" relationships).

e.g. synonym sets containing "good":

↳ not good for anything but has a lot of basic things.

```
from nltk.corpus import wordnet as wn
poses = { 'n':'noun', 'v':'verb', 's':'adj (s)', 'a':'adj', 'r':'adv'}
for synset in wn.synsets("good"):
    print("{}: {}".format(poses[synset.pos()],
            ", ".join([l.name() for l in synset.lemmas()])))
```

tell me about synonym sets good participates in.

```
noun: good
noun: good, goodness
noun: good, goodness
noun: commodity, trade_good, good
adj: good
adj (sat): full, good
adj: good
adj (sat): estimable, good, honorable, respectable
adj (sat): beneficial, good
adj (sat): good
adj (sat): good, just, upright
…
adverb: well, good
adverb: thoroughly, soundly, good
```

person who is respectable

(good in sense of beneficial)

e.g. hypernyms of "panda":

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

```
[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')]
```

stuff you get as a result of wordnet.

Problematic in practice to use: tries to make all fine-grained differences between senses a human being can barely understand the difference btw them.

Problems with resources like WordNet

never worked that well

- Great as a resource but missing nuance
  - e.g. "proficient" is listed as a synonym for "good". This is only correct in some contexts. (proficient has some more nuance)
- Missing new meanings of words (incomplete)
  - e.g., wicked, badass, nifty, wizard, genius, ninja, bombest
  - Impossible to keep up-to-date!
- Subjective
- Requires human labor to create and adapt
- Can't compute accurate word similarity →

Basic thing you would like to do with words:

- Wordnet doesn't do that well, because it has just these fixed discrete syn sets.
  - Words in syn. set, are sort of a syn. but not exactly the same meaning. If not in a syn. set, can't resemble partial resemblance of meaning for them.

For ex: good ad marvelous not in the same set, but you like resemble some sort of thing their's have in common.

Build from Traditional NLP

↳ up until 2012-2013 change as ppl. use neural net style representations for language processing.

Representing words as discrete symbols

In traditional NLP, we regard words as discrete symbols:
hotel, conference, motel – a localist representation

→ for any concept there is one particular place which is the word hotel or the word motel.

→ categorical variable coded as having different lvls. of variables, each index is a diff. word.
- also called 1-hot vectors.

Means one 1, the rest 0s

Words can be represented by one-hot vectors:

motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
hotel = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]

Draw back: ① Minor problem

Vector dimension = number of words in vocabulary (e.g., 500,000)

So we very big vectors.

→ in english infinite words, make bigger words by adding more stuff into it.

② Much bigger problem:

Problem with words as discrete symbols

Precisely we want to understand relationships between meaning of words all the time.

Example: in web search, if user searches for "Seattle motel", we would like to match documents containing "Seattle hotel".

But: (in 1-hot vectors)

motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]

hotel = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]

These two vectors are orthogonal.

There is no natural notion of **similarity** for one-hot vectors!

So you get nowhere.

Solution:

(word similarity tables. $|V| \times |V|$ = 2.5 trillion very big)

- Could try to rely on WordNet's list of synonyms to get similarity?
  - But it is well-known to fail badly: incompleteness, etc.
- **Instead: learn to encode similarity in the vectors themselves**

(represent words as vectors in such way just the representation of a word gives you their similarity with no further work.)

Representing words by their context

Another idea for representing meaning of words.

- Distributional semantics: **A word's meaning is given by the words that frequently appear close-by**

→ Understand the meaning of a word by looking at that context in which it appears.

  - *"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!

- When a word *w* appears in a text, its **context** is the set of words that appear nearby (within a fixed-size window).

- Use the many contexts of *w* to build up a representation of *w*

find sentences where banking is used.

…government debt problems turning into	***banking***	*crises as happened in 2009…*
…saying that Europe needs unified	***banking***	*regulation to replace the hodgepodge…*
…India has just given its	***banking***	*system a shot in the arm…*

→ very useful idea that does a great job at capturing meaning.

These context words will represent ***banking***

(these are meaning of banking word, or context in which it is used.)

Word vectors

Rather than our old localist representation, we are going to use what we call a distributed representation.

- now a smallish dense vector where all no.s are non-zero.

We will build a dense vector for each word, chosen so that it is similar to vectors of words that appear in similar contexts

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

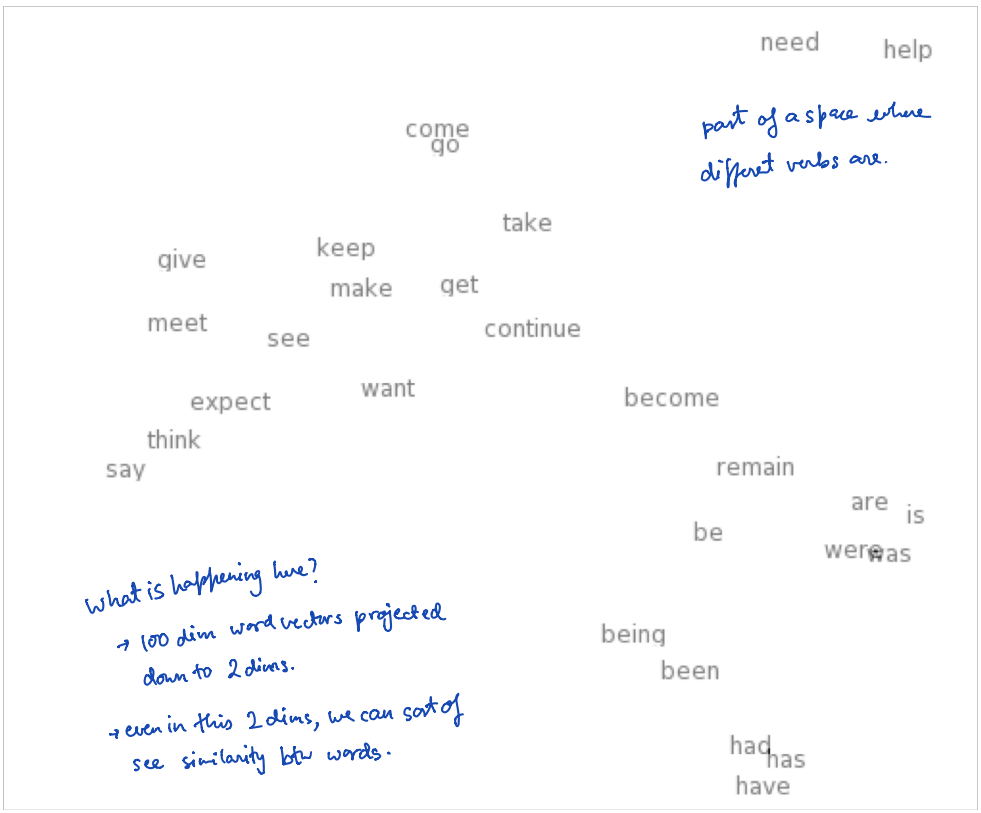
- The meaning of banking is going to be distributed over this vector.
- in practice, we use a large dimension like 50, 500 etc. but orders of magnitude smaller compared to length 500,000 vector.

Note: word vectors are sometimes called word embeddings or word representations. They are a distributed representation.

Word meaning as a neural word vector – visualization

→ Since each word has a vector representation, we have a vector space in which we can place all of the words.

$$expect = \begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \\ 0.487 \end{pmatrix}$$



part of a space where different verbs are.

What is happening here?

→ 100 dim word vectors projected down to 2 dims.

→ even in this 2 dims, we can sort of see similarity btw words.

3. Word2vec: Overview (Thing that had the biggest impact of turning NLP in this neural nets direction.)

Word2vec (Mikolov et al. 2013) is a framework for learning word vectors

- not the 1st work in learning distributed representation
- but this very simple and very scalable way of learning vector representations of words.

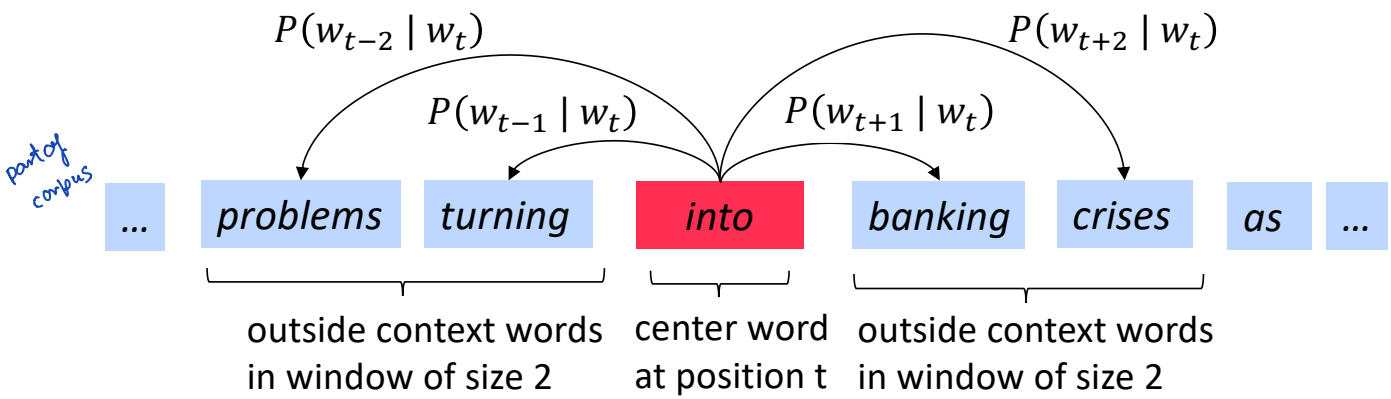
Idea:

- We have a large corpus of text (a large pile of text / (Latin word for body, a body of text:) / start with continuous text, actual sentences.)
- Every word in a fixed vocabulary is represented by a vector (and start those vectors off as random vectors.)
- Iterative algo.
- Go through each position *t* in the text, which has a center word *c* and context ("outside") words *o*
- Use the similarity of the word vectors for *c* and *o* to calculate the probability of *o* given *c* (or vice versa)
- Keep adjusting the word vectors to maximize this probability

- go through each position in text, here's a word in the text, look at the words around it.
  - → And the meaning of word is it's context of use.
  - → Want the representation of the word in middle, to be able to predict words around it.
  - → Achieve that by moving the position of word vectors repeat that.

Word2Vec Overview

- Example windows and process for computing $P(w_{t+j} \mid w_t)$

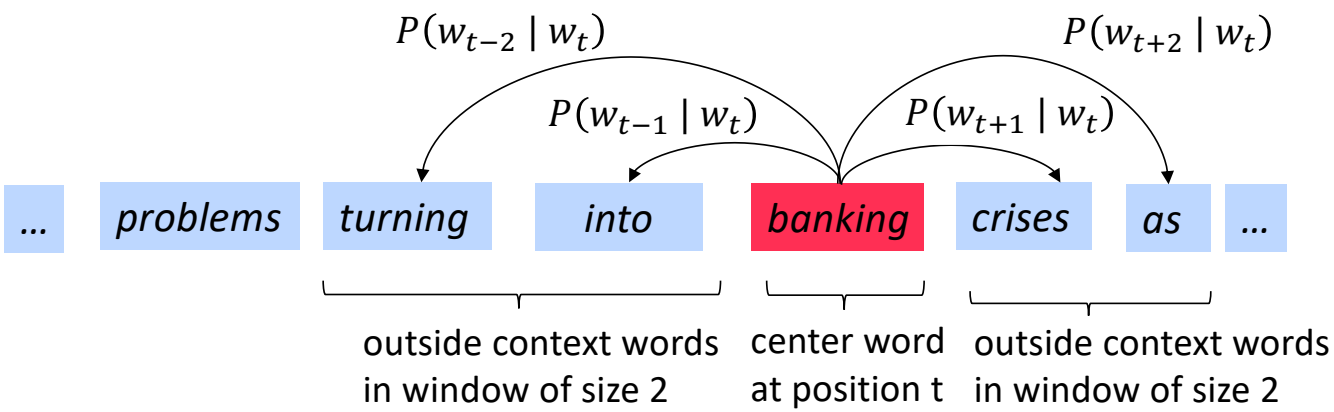


Representation of "into": → can be used to predict what words appear in the context of "into".
→ that's the meaning of "into".

Word2Vec Overview

(dealing with "into", go to next word)

- Example windows and process for computing $P(w_{t+j} \mid w_t)$



Word2vec: objective function

T words (long list of words)

For each position $t = 1, \ldots, T$, predict context words within a window of fixed size m, given center word w_j.

Likelihood = $L(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \le j \le m \\ j \neq 0}} P(w_{t+j} \mid w_t; \theta)$

θ is all variables to be optimized

parameters of model

go through all the words

choose some fixed sized window, and predict the 10 words around our centre word; & predict in that sense of trying to predict that word given the centre word. That's our probability model.

if all multiply all those things together: model likelihood, how good job it does at predicting words around every word.

only parameters are the vector representations we give to those words; Model has no other parameters to it.

sometimes called *cost* or *loss* function

The objective function $J(\theta)$ is the (average) negative log likelihood:

$$J(\theta) = -\frac{1}{T} \log L(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \le j \le m \\ j \neq 0}} \log P(w_{t+j} \mid w_t; \theta)$$

minimise rather than maximise.

avg. goodness (keeps scales not dependent on size of corpus.)

Minimizing objective function ⟺ Maximizing predictive accuracy

- We are representating a word with a vector in a vector space, and that representation of it is it's meaning.
- With that we are going to be able to predict what other words occur in a way.

Word2vec: objective function

- We want to minimize the objective function:

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{\substack{-m\le j\le m \\ j\neq 0}} \log P\left(w_{t+j} \mid w_t; \theta\right)$$

- Question: How to calculate $P\left(w_{t+j} \mid w_t; \theta\right)$?
- Answer: We will *use two* vectors per word *w*: (makes it simpler)
  - (1) v_w when *w* is a center word (predicting other words)
  - (2) u_w when *w* is a context word
- Then for a center word *c* and a context word *o*:

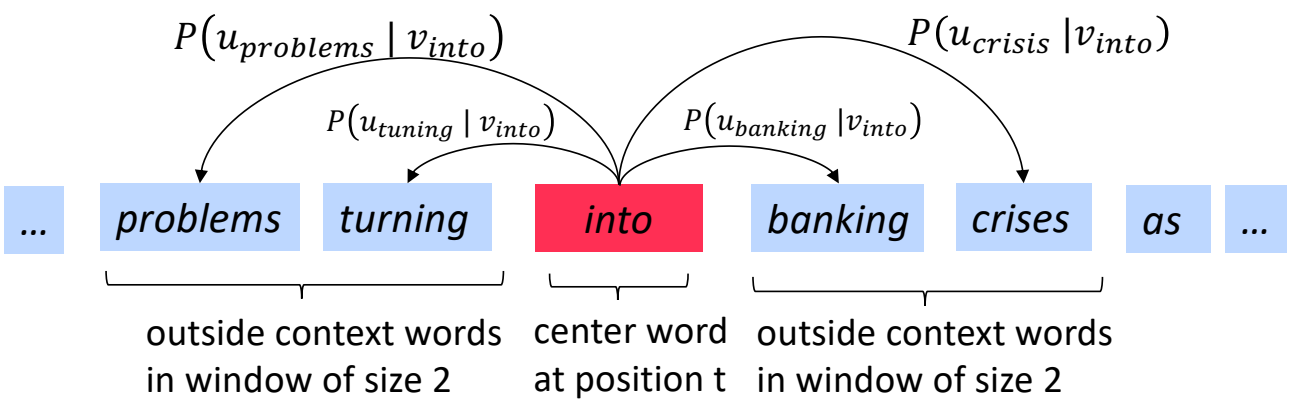
probability of word in context given center word.

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

Word2Vec Overview with Vectors

- Example windows and process for computing $P(w_{t+j} \mid w_t)$
- $P(u_{problems} \mid v_{into})$ short for $P(problems \mid into\ ; u_{problems}, v_{into}, \theta)$

still in exactly the same situation: predict probabilities of words given our center word.



Word2vec: prediction function

(shape you see in DL with categorical stuff.)

Exponentiation makes anything positive

(dot product might be +ve/-ve)

Calculates similarity btw words, similarity meaning vectors looking the same.

Dot product compares similarity of *o* and *c*.

$u^T v = u.v = \sum_{i=1}^{n} u_i v_i$

Larger dot product = larger probability if same sign

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

sum this prob. for every word in vocabulary.

Normalize over entire vocabulary to give probability distribution

Two parts of exp( ) normalize & give a softmax distribution.

- This is an example of the **softmax function** $\mathbb{R}^n \rightarrow \mathbb{R}^n$

  → map any nos to a prob. distribution.

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)} = p_i$$

- The softmax function maps arbitrary values x_i to a probability distribution p_i
  - "max" because amplifies probability of largest x_i
  - "soft" because still assigns some probability to smaller x_i

    → exp → makes big nos very big. So put mass where the max is.

    hardmax - everything else is 0.
  - Frequently used in Deep Learning

We have a loss fn with prob. model on inside that we can build,

→ so what we want to be able to do: • have our vector representations of words around, so they are good at predicting what words occur in context of other words.

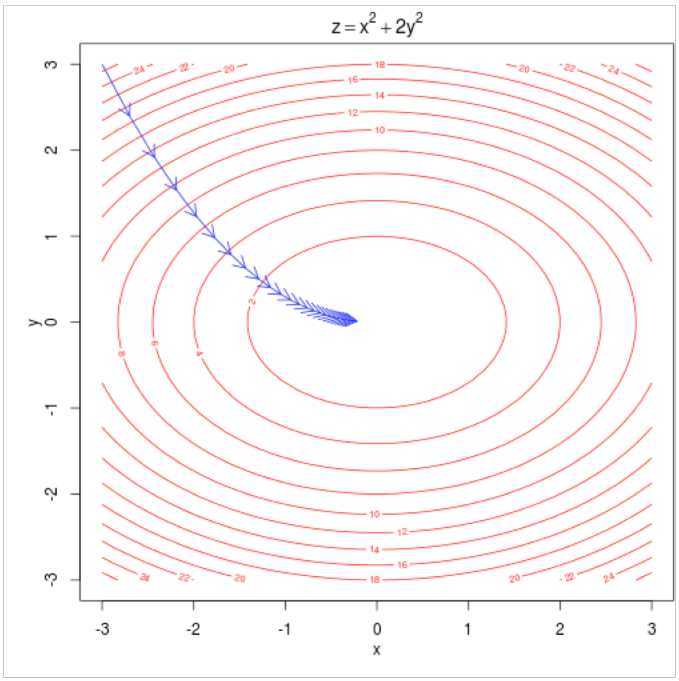
Training a model by optimizing parameters

To train a model, we adjust parameters to minimize a loss

E.g., below, for a simple convex function over two parameters

Contour lines show levels of objective function

walking down the gradient so as to minimize this fn. & we can find good representations for these words.



just 2 dim for picture.

To train the model: Compute all vector gradients!

- Recall: θ represents **all** model parameters, in one long vector
- In our case with *d*-dimensional vectors and *V*-many words:

$$\theta = \begin{bmatrix} v_{aardvark} \\ v_a \\ \vdots \\ v_{zebra} \\ u_{aardvark} \\ u_a \\ \vdots \\ u_{zebra} \end{bmatrix} \in \mathbb{R}^{2dV}$$

the only parameters this model has are the vector space representations of word.

- Remember: every word has two vectors
- We optimize these parameters by walking down the gradient

4. Word2vec derivations of gradient

- Whiteboard – see video if you're not in class ;)
- The basic Lego piece
- Useful basics: $\frac{\partial \mathbf{x}^T \mathbf{a}}{\partial \mathbf{x}} = \frac{\partial \mathbf{a}^T \mathbf{x}}{\partial \mathbf{x}} = \mathbf{a}$
- If in doubt: write out with indices

- Chain rule! If $y = f(u)$ and $u = g(x)$, i.e. $y = f(g(x))$, then:

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

$$\max J'(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \le j \le m \\ j \neq 0}} p(w_{t+j} \mid w_t ; \theta)$$

$$\min J(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \le j \le m \\ j \neq 0}} \log p(w'_{t+j} \mid w_t)$$

change!

$$p(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w=1}^{v} \exp(u_w^T v_c)}$$

Parameters are our word vectors.

numerator / denominator

$$\frac{\partial}{\partial v_c} \log \frac{\exp(u_o^T v_c)}{\sum_{w=1}^{v} \exp(u_w^T v_c)} = \frac{\partial}{\partial v_c} \left[\log \exp(u_o^T v_c) - \log \sum_{w=1}^{v} \exp(u_o^T v_c) \right]$$

$$= \frac{\partial}{\partial v_c} u_o^T v_c - \frac{\partial}{\partial v_c} \log \sum_{w=1}^{v} \exp(u_o^T v_c)$$

still a vector

$= u_o$

f → chain rule

body of f: $z(v_c)$

sum of z

first dimension

$$\frac{\partial}{\partial (v_c)_1} u_{o_1} v_{c_1} + u_{o_2} v_{c_2} \dots + u_{o_{100}} v_{c_{100}} = u_{o_1}$$

all dims $= [u_{o_1}, u_{o_2}, u_{o_3} \dots]$

$$\frac{1}{\sum_{w=1}^{v} \exp(u_o^T v_c)} \cdot \frac{\partial}{\partial v_c} \sum_{x=1}^{v} \exp(u_x^T v_c)$$

more derivative inside the sum.

$$\sum_{x=1}^{v} \frac{\partial}{\partial v_c} \exp(u_x^T v_c) \quad \frac{1}{f} \quad \frac{1}{z(v_c)}$$

chain rule

$$\sum_{x=1}^{v} \exp(u_x^T v_c) \times \frac{d}{\partial v_c} u_x^T v_c$$

$$\sum_{x=1}^{v} \exp(u_x^T v_c)\, u_x$$

Put it together

$$\frac{\partial}{\partial v_c} \log p(o|c) = u_o - \frac{\sum_{x=1}^{v} \exp(u_x^T v_c)\, u_x}{\sum_{w=1}^{v} \exp(u_w^T v_c)}$$

gives us our slope in multiple dim. space.

Rearrange.

$$= u_o - \sum_{x=1}^{v} \frac{\exp(u_x^T v_c)}{\sum_{w=1}^{v} \exp(u_w^T v_c)}\, u_x$$

$p(x|c)$

→ Rediscover the same form we use to predict prob. distribution of words

$$= u_o - \sum_{x=1}^{v} p(x|c).u_x$$

→ Difference btw actual context word & expected context word; that difference gives the slope we should be walking in order to improve our model's ability to predict.

observed representation of context word

Subtracting from that what our model thinks context should like.

Formally an expectation. Representation of each word multiplied by the probability in the current model.

Sort of an expected context word in current model.

Chain Rule

- Chain rule! If $y = f(u)$ and $u = g(x)$, i.e. $y = f(g(x))$, then:

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx} = \frac{df(u)}{du}\frac{dg(x)}{dx}$$

- Simple example: $\frac{dy}{dx} = \frac{d}{dx}5(x^3+7)^4$

$$y = f(u) = 5u^4 \qquad u = g(x) = x^3 + 7$$

$$\frac{dy}{du} = 20u^3 \qquad \frac{du}{dx} = 3x^2$$

$$\frac{dy}{dx} = 20(x^3+7)^3.3x^2$$

Interactive Whiteboard Session!

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

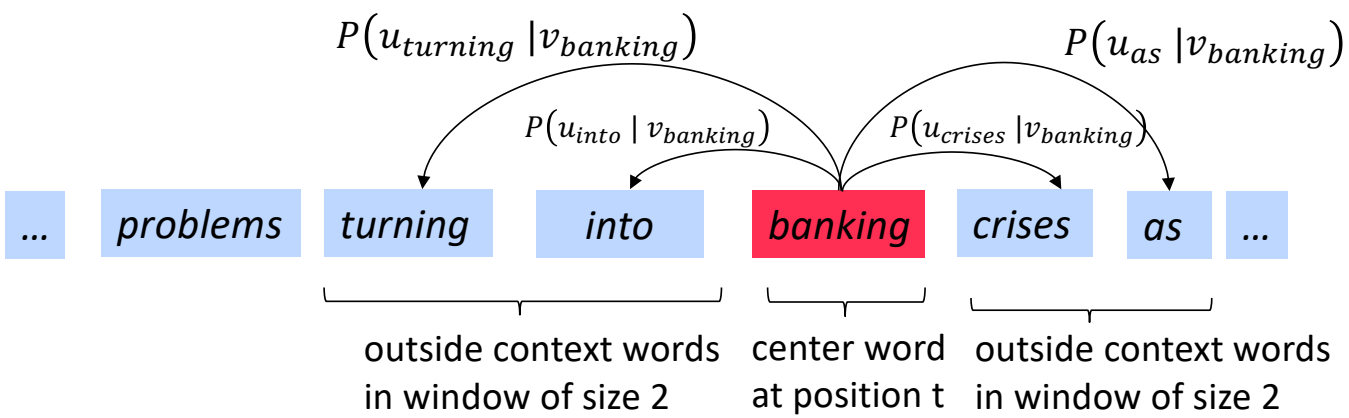
Let's derive gradient for center word together
For one example window and one example outside word:

$$\log p(o|c) = \log \frac{\exp({u_o}^T v_c)}{\sum_{w=1}^{V} \exp({u_w}^T v_c)}$$

You then also need the gradient for context words (it's similar; left for homework). That's all of the parameters θ here.

Calculating all gradients!

- We went through gradient for each center vector *v* in a window
- We also need gradients for outside vectors *u*
  - Derive at home!
- Generally in each window we will compute updates for all parameters that are being used in that window. For example:



Word2vec: More details

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

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

This lecture so far: **Skip-gram model**

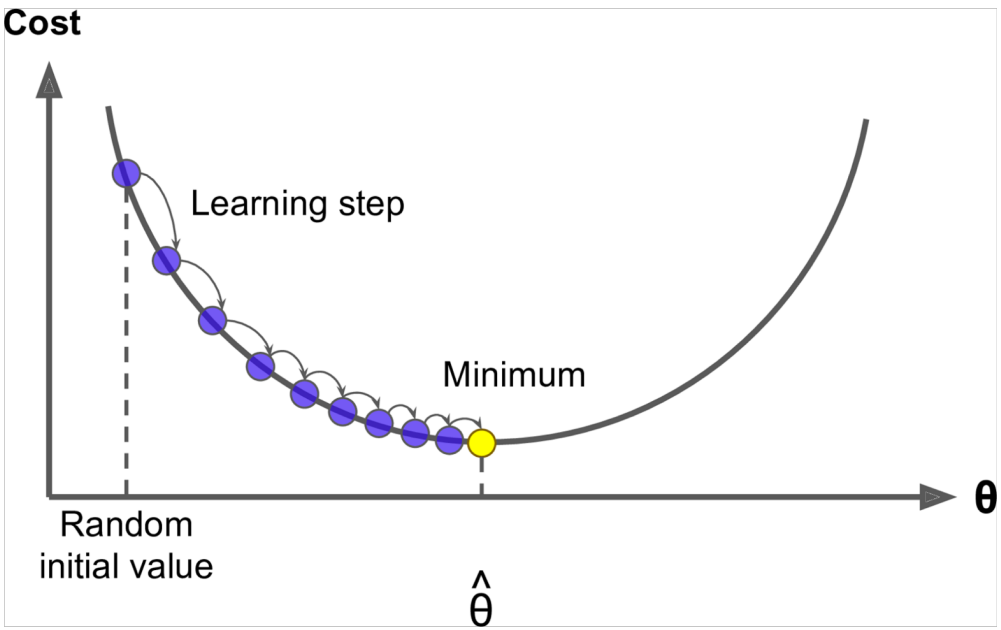
Additional efficiency in training:

1. Negative sampling

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

5. Optimization: Gradient Descent

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



Note: Our objectives may not be convex like this :(

Gradient Descent

- Update equation (in matrix notation):

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

α = *step size* or *learning rate*

- Update equation (for single parameter):

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

- Algorithm:

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

Stochastic Gradient Descent

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

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

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

Lecture Plan

1. The course (10 mins)
2. Human language and word meaning (15 mins)
3. Word2vec introduction (15 mins)
4. Word2vec objective function gradients (25 mins)
5. Optimization basics (5 mins)
6. Looking at word vectors (10 mins or less)