

# Natural Language Processing with Deep Learning

## CS224N/Ling284



**Christopher Manning** and Richard Socher  
Lecture 2: Word Vectors



# Lecture Plan

1. Word meaning (15 mins)
2. Word2vec introduction (20 mins)
3. Research highlight (Danqi) (5 mins)
4. Word2vec objective function gradients (25 mins)
5. Optimization refresher (10 mins)

Fire alarm allowance: 5 mins

# 1. 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.

Commonest linguistic way of thinking of meaning:

- signifier  $\Leftrightarrow$  signified (idea or thing) = denotation

# How do we have usable meaning in a computer?

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

```
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')]
```

(here, for *good*):

```
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 a 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

It is a **localist** representation

# From symbolic to distributed representations

Its problem, e.g., for web search

- If user searches for [Dell notebook battery size], we would like to match documents with “Dell laptop battery capacity”
- If user searches for [Seattle motel], we would like to match documents containing “Seattle hotel”

But

$$\begin{array}{l} \text{motel} \quad [0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0]^T \\ \text{hotel} \quad [0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0]^T = 0 \end{array}$$

Our query and document vectors are **orthogonal**

There is no natural notion of similarity in a set of one-hot vectors

Could deal with similarity separately;

instead we explore a direct approach where vectors encode it

# 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* ↗



# Word meaning is defined in terms of vectors

We will build a dense vector for each word type, chosen so that it is good at predicting other words appearing in its context

... those other words also being represented by vectors ... it all gets a bit recursive

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

# Basic idea of learning neural network word embeddings

We define a model that aims to predict between a center word  $w_t$  and context words in terms of word vectors

$$p(\text{context} | w_t) = \dots$$

which has a loss function, e.g.,

$$J = 1 - p(w_{-t} | w_t)$$

We look at many positions  $t$  in a big language corpus

We keep adjusting the vector representations of words to minimize this loss

# Directly learning 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 (almost) from Scratch (Collobert & Weston, 2008)
- A recent, even simpler and faster model:  
word2vec (Mikolov et al. 2013) → intro now

## 2. Main idea of word2vec

Predict between every word and its context words!

Two algorithms

1. **Skip-grams (SG)**

Predict context words given target (position independent)

2. **Continuous Bag of Words (CBOW)**

Predict target word from bag-of-words context

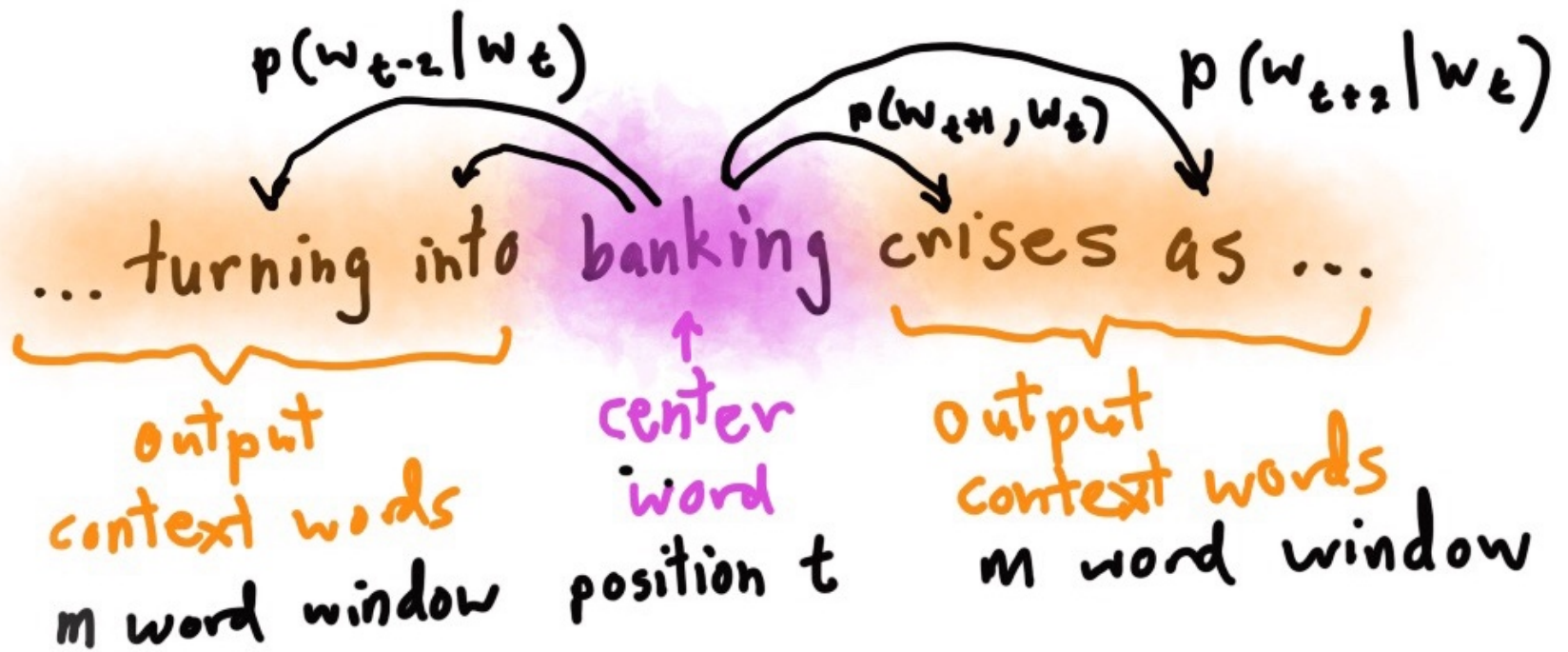
Two (moderately efficient) training methods

1. Hierarchical softmax

2. Negative sampling

**Naïve softmax**

# Skip-gram prediction



## Details of word2vec

For each word  $t = 1 \dots T$ , predict surrounding words in a window of “radius”  $m$  of every word.

Objective function: Maximize the probability of any context word given the current center word:

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

Negative  
Log  
Likelihood

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

Where  $\theta$  represents all variables we will optimize

# The objective function – details

- Terminology: Loss function = cost function = objective function
- Usual loss for probability distribution: Cross-entropy loss
- With one-hot  $w_{t+j}$  target, the only term left is the negative log probability of the true class
- More on this later...

## Details of Word2Vec

Predict surrounding words in a window of radius  $m$  of every word

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

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

一个单词用两个Vector表示  
在数学上更为简单

where  $o$  is the outside (or output) word index,  $c$  is the center word index,  $v_c$  and  $u_o$  are “center” and “outside” vectors of indices  $c$  and  $o$

**Softmax** using word  $c$  to obtain probability of word  $o$



# Dot products

Dot product

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

Bigger if  $u$  and  $v$  are more similar!

Iterate over  $w=1 \dots W$ :  $u_w^T v$  means:

Work out how similar each word is to  $v$ !

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

# Softmax function: Standard map from $\mathbb{R}^V$ to a probability distribution

*Exponentiate to make positive*

Softmax

$$e^{u_i}$$

*Normalize to give probability*

$$p_i =$$

$$\frac{e^{u_i}}{\sum_j e^{u_j}}$$

# Skipgram

$V \times 1$   $d \times V$   $d \times 1$   
 $w_e$   $W$   $v_c = Ww_e$

0	0.2	0.2
0	-1.4	-1.4
0	0.3	0.3
0	-0.1	-0.1
1	0.1	0.1
0	0.5	0.5
0		

↑  
 one hot  
 word  
 symbol  
 ↑  
 word

↑  
 Looks up  
 column of  
 word embedding  
 matrix as  
 representation  
 of center word

↑  
 Output  
 word  
 representation

$V \times d$   
 $W'$

$V \times 1$   $V \times 1$   
 $W'v_c = [u_x^T v_c]$   $p(x|c) = \text{softmax}(u_x^T v_c)$

$\begin{bmatrix} 6.2 \\ 6.3 \\ 0.1 \\ -6.7 \\ -0.2 \\ 0.1 \\ 0.7 \end{bmatrix}$

softmax  
 →

$\begin{bmatrix} 0.07 \\ 6.1 \\ 6.05 \\ 6.01 \\ 0.02 \\ 0.05 \\ 0.7 \end{bmatrix}$

$V \times 1$   
 Truth

$\begin{bmatrix} 0 \\ 6 \\ 6 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

$w_{t-3}$

Softmax

$p_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$

$\begin{bmatrix} 6.2 \\ 6.3 \\ 0.1 \\ -6.7 \\ -0.2 \\ 0.1 \\ 0.7 \end{bmatrix}$

softmax  
 →

$\begin{bmatrix} 0.07 \\ 6.1 \\ 6.05 \\ 6.01 \\ 0.02 \\ 0.05 \\ 0.7 \end{bmatrix}$

$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

$w_{t-2}$

Actual  
 context  
 words

$\begin{bmatrix} 6.2 \\ 6.3 \\ 0.1 \\ -6.7 \\ -0.2 \\ 0.1 \\ 0.7 \end{bmatrix}$

softmax  
 →

$\begin{bmatrix} 0.07 \\ 6.1 \\ 6.05 \\ 6.01 \\ 0.02 \\ 0.05 \\ 0.7 \end{bmatrix}$

$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$

$w_{t-1}$

# To train the model: Compute **all** vector gradients!

- We often define the set of **all** parameters in a model in terms of one long vector  $\theta$

- In our case with  $d$ -dimensional vector and  $V$  many words:

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

- We then optimize these parameters

Note: Every word has two vectors! Makes it simpler!

## 4. 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:

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

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

$$y = f(u) = 5u^4$$

$$u = g(x) = x^3 + 7$$

$$\frac{dy}{du} = 20u^3$$

$$\frac{du}{dx} = 3x^2$$

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

# Interactive Whiteboard Session!

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

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 params  $\theta$  here.



## Objective Function

$$\text{Maximize } J'(\theta) = \prod_{t=1}^T \prod_{\substack{-m \leq j \leq m \\ j \neq 0}} p(w'_{t+j} | w_t; \theta)$$

Or minimize  
neg. log  
likelihood

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

[negate to minimize;  
log is monotone]

↑  
text  
length

↑  
window  
size

where

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

word IDs ↗

We now take derivatives to work out minimum

Each word type  
(vocab entry)  
has two word  
representations:  
as center word  
and context word



$$\frac{\partial}{\partial v_c} \log \frac{\exp(u_0^T v_c)}{\sum_{w=1}^V \exp(u_w^T v_c)}$$

$$= \underbrace{\frac{\partial}{\partial v_c} \log \exp(u_0^T v_c)}_{\textcircled{1}} - \underbrace{\frac{\partial}{\partial v_c} \log \sum_{w=1}^V \exp(u_w^T v_c)}_{\textcircled{2}}$$

$$\textcircled{1} \quad \frac{\partial}{\partial v_c} \underbrace{\log \exp(u_0^T v_c)}_{\text{inverses}} = \frac{\partial}{\partial v_c} u_0^T v_c = u_0$$

Vector!  
Not high  
school  
single  
variable  
calculus

You can do things one variable at a time,  
and this may be helpful when things  
get gnarly.

$$\forall j \quad \frac{\partial}{\partial (v_c)_j} u_0^T v_c = \frac{\partial}{\partial (v_c)_j} \sum_{i=1}^d (u_0)_i (v_c)_i$$

$$= (u_0)_j$$

Each term is zero except when  $i=j$

$$\textcircled{2} \frac{\partial}{\partial v_c} \log \underbrace{\sum_{w=1}^v \exp(u_w^T v_c)}_{f \quad z = g(v_c)}$$

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

$$\frac{\partial}{\partial v_c} f(\overbrace{g(v_c)}^z) = \frac{\partial f}{\partial z} \cdot$$

$$\frac{\partial z}{\partial v_c}$$

Use chain rule

$$= \frac{1}{\sum_{w=1}^v \exp(u_w^T v_c)} \cdot \left( \sum_{x=1}^v \frac{\partial}{\partial v_c} \underbrace{\exp(u_x^T v_c)}_{f \quad z = g(v_c)} \right)$$

Move deriv  
inside sum

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

chain  
rule

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

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

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

Distribute  
term  
across sum

$$= u_o - \underbrace{\sum_{x=1}^V p(x|c)}_{\text{this is an expectation: average over all context vectors weighted by their probability}} u_x$$

= observed - expected

This is just the derivatives for the center vector parameters

Also need derivatives for output vector parameters

(they're similar)

Then we have derivative w.r.t. all parameters and can minimize

# 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, window size  $m = 1$ , sentence:  
“We like learning a lot”
- First window computes gradients for:
  - internal vector  $v_{\text{like}}$  and external vectors  $u_{\text{We}}$  and  $u_{\text{learning}}$
- Next window in that sentence?

## 5. Cost/Objective functions

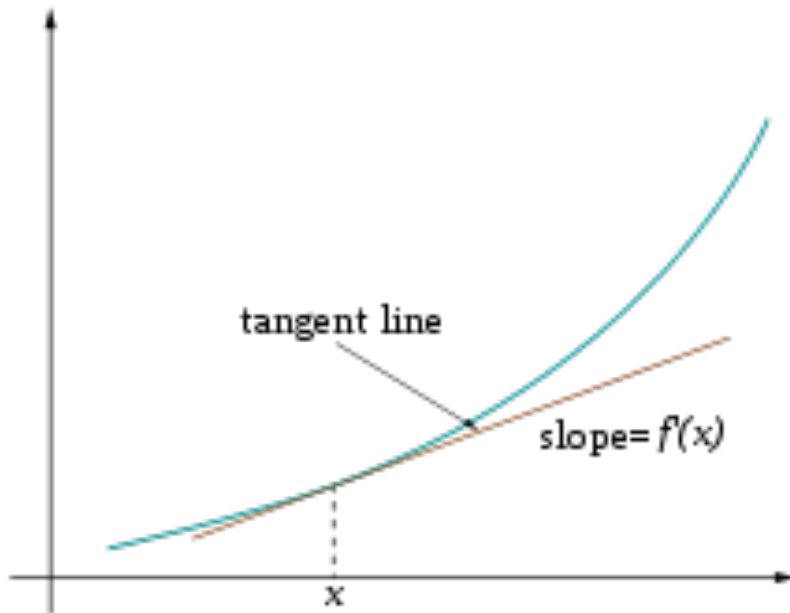
We will optimize (maximize or minimize) our objective/cost functions

For now: minimize → gradient descent

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)
```

**Subtracting a fraction of the gradient moves you towards the minimum!**

# 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  $\theta$  :

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

- With step size  $\alpha$
- 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

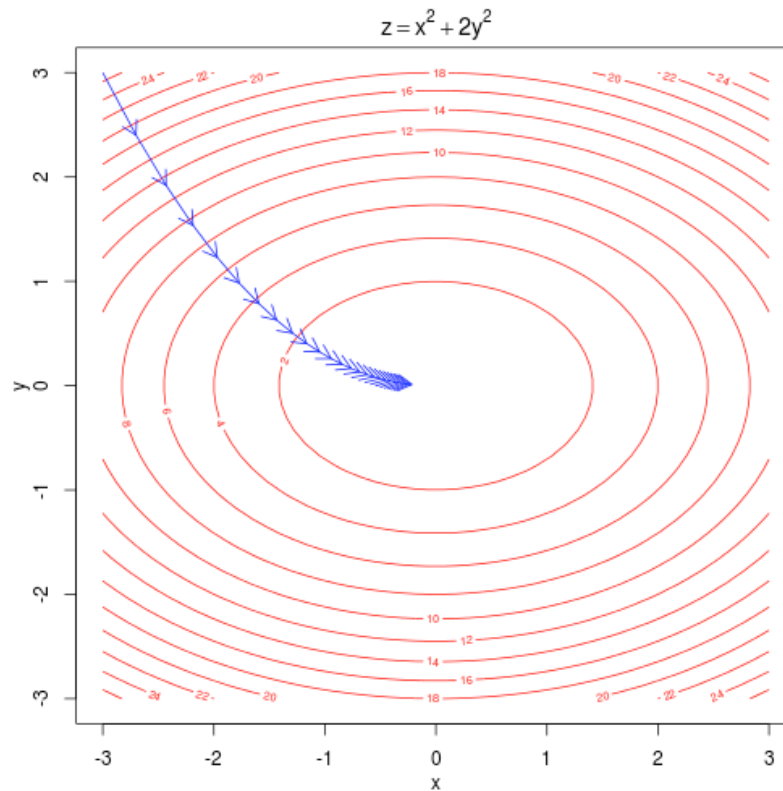
$$\theta^{new} = \theta^{old} - \alpha \nabla_{\theta} J(\theta)$$

```
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





# 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
```