Natural Language Processing

AI51701/CSE71001

Lecture 4

09/07/2023

Instructor: Taehwan Kim

Word (review)

How should we evaluate word vectors?

- WordSim353 (Finkelstein et al., 2002)

word pair		similarity
journey	voyage	9.3
king	queen	8.6
computer	software	8.5
law	lawyer	8.4
forest	graveyard	1.9
rooster	voyage	0.6

How should we evaluate word vectors?

- SimLex-999 (Hill et al., 2014)

word pair		similarity
insane	crazy	9.6
attorney	lawyer	9.4
author	creator	8.0
diet	apple	1.2
new	ancient	0.2

measures **paraphrastic** similarity:

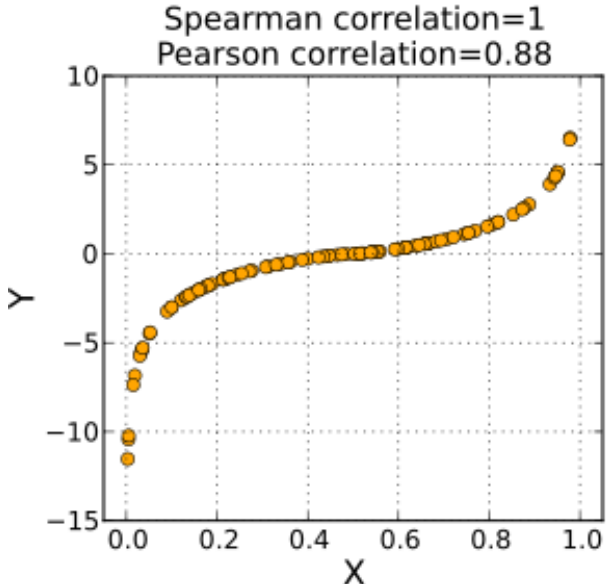
two words are “similar” if they have similar meanings

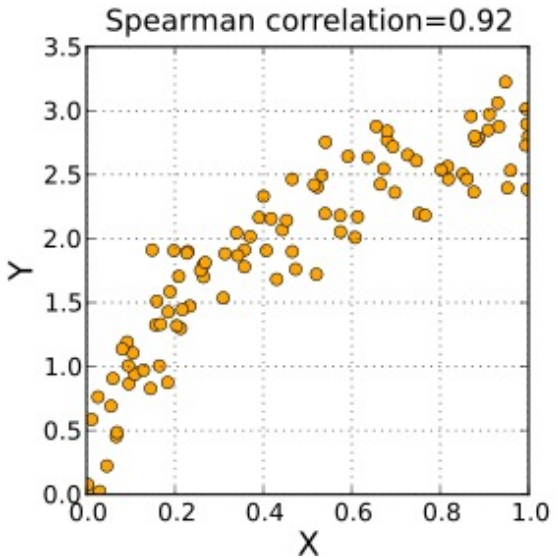
How should we evaluate word vectors?

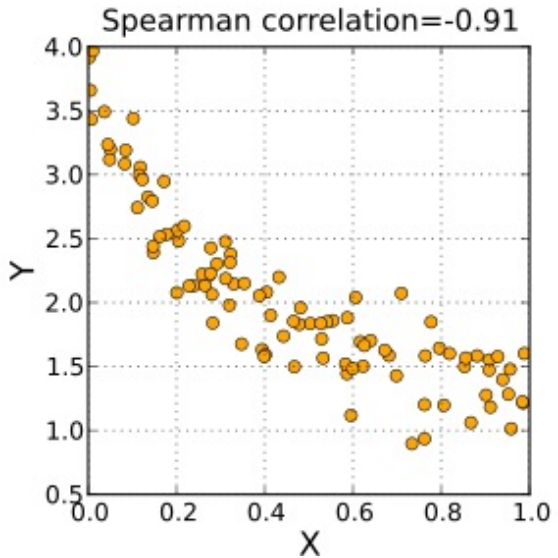
- There are many word similarity datasets
- Some focus on topical **relatedness**, others focus on similarity in **meaning**

Evaluation Metrics for Word Similarity

- Spearman rank correlation coefficient
- Measures correlation between two variables:
  - variable 1: human-annotated similarities for word pairs
  - variable 2: cosine similarities computed with your word vectors for the same word pairs





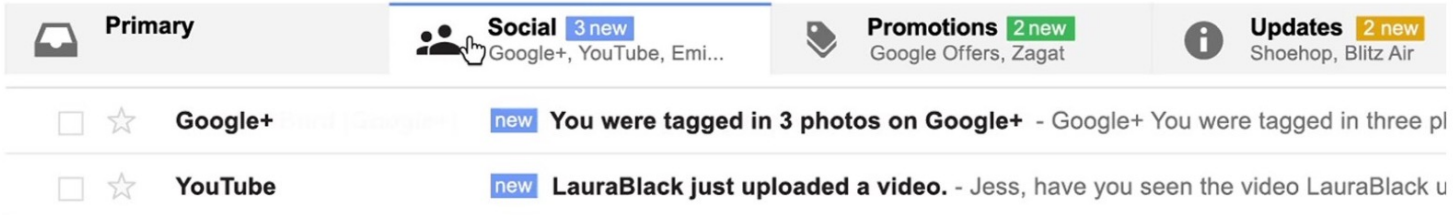


From Wikipedia

Text Classification

Text Classification

- ❑ Simplest user-facing NLP application
- ❑ Email (spam, priority, categories):



- ❑ Sentiment:

- ❑ Topic classification
- ❑ Others?

Text Classification

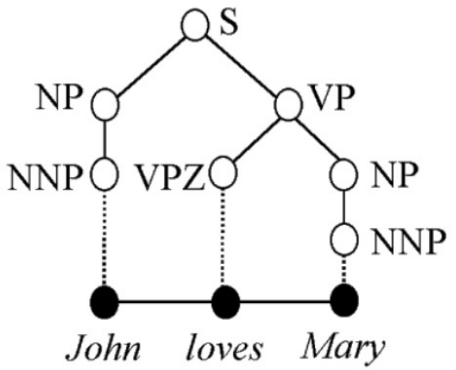
- **Datasets**
- Classification
  - Modeling
  - Inference
  - Learning

Annotation

- Supervised machine learning needs labeled datasets, where labels are called **ground truth**
- In NLP, labels are annotations provided by humans
- There is always some disagreement among annotators, even for simple tasks
- These annotations are called a **gold standard**, not ground truth

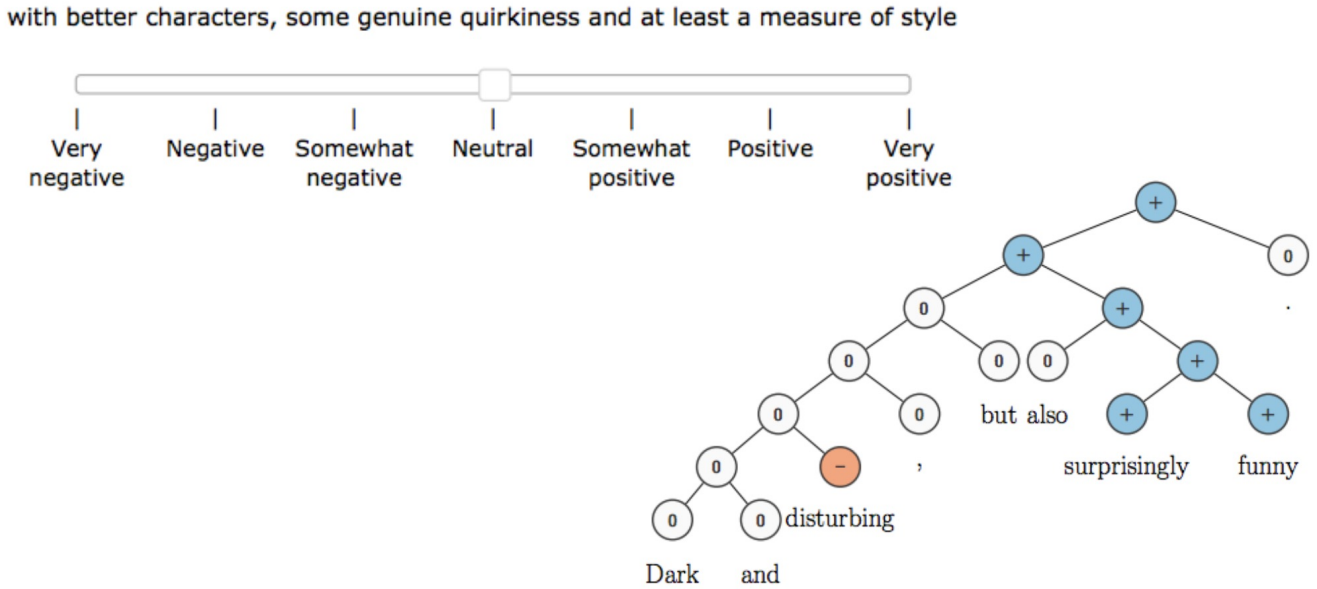
How are NLP datasets developed?

- Paid, trained human annotators
  - traditional approach
  - researchers write annotation guidelines, recruit & pay annotators (often linguists)
  - more consistent annotations, but costly to scale
  - e.g., Penn Treebank (1993)
    - 1 million words, mostly Wall Street Journal, annotated with part-of-speech tags and syntactic parse trees



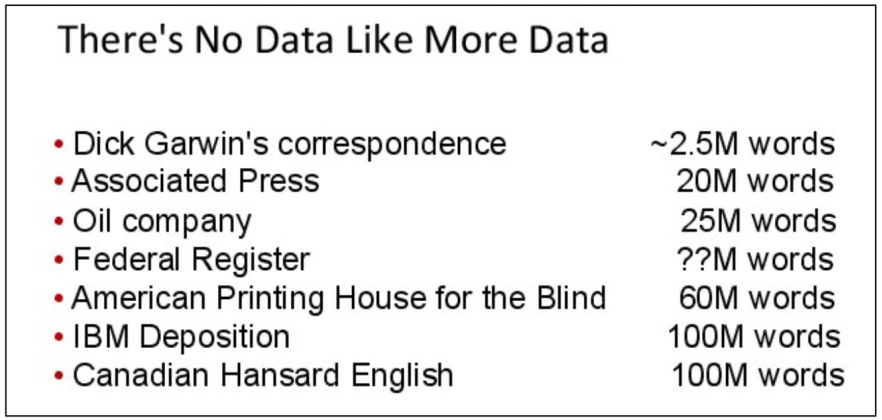
Crowdsourcing

- Crowdsourcing (e.g., Amazon Mechanical Turk)
  - more recent trend
  - can't really train annotators, but easier to get multiple annotations for each input (which can then be averaged)
  - e.g., Stanford Sentiment Treebank



Naturally-occurring annotation

- Long history: used by IBM for speech recognition



credit: Brown & Mercer, 20 Years of Bitext Workshop, 2013

- How might you find naturally-occurring data for:
  - conversational agents
  - summarization
  - coreference resolution

Text Classification

- Datasets
- **Classification**
  - Modeling
  - Inference
  - Learning

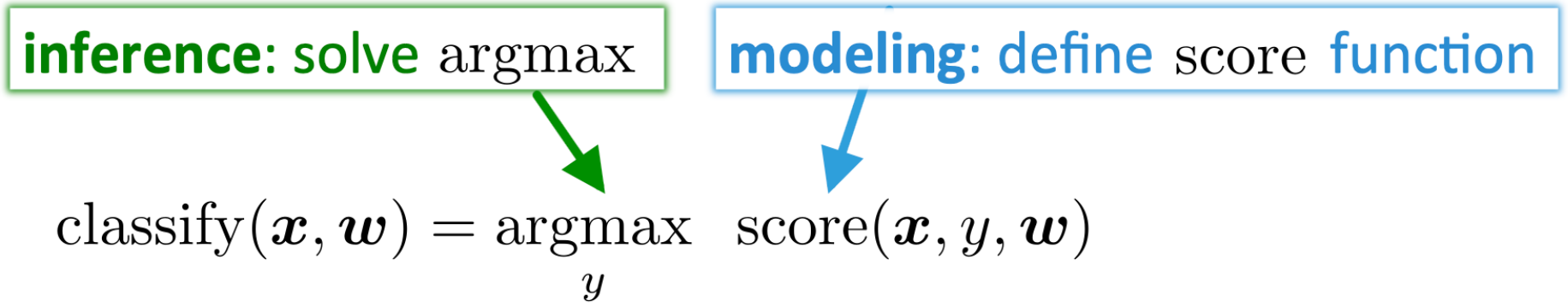
Modeling, Inference, Learning

modeling: define score function

$$\text{classify}(\boldsymbol{x}, \boldsymbol{w}) = \underset{y}{\text{argmax}} \quad \text{score}(\boldsymbol{x}, y, \boldsymbol{w})$$

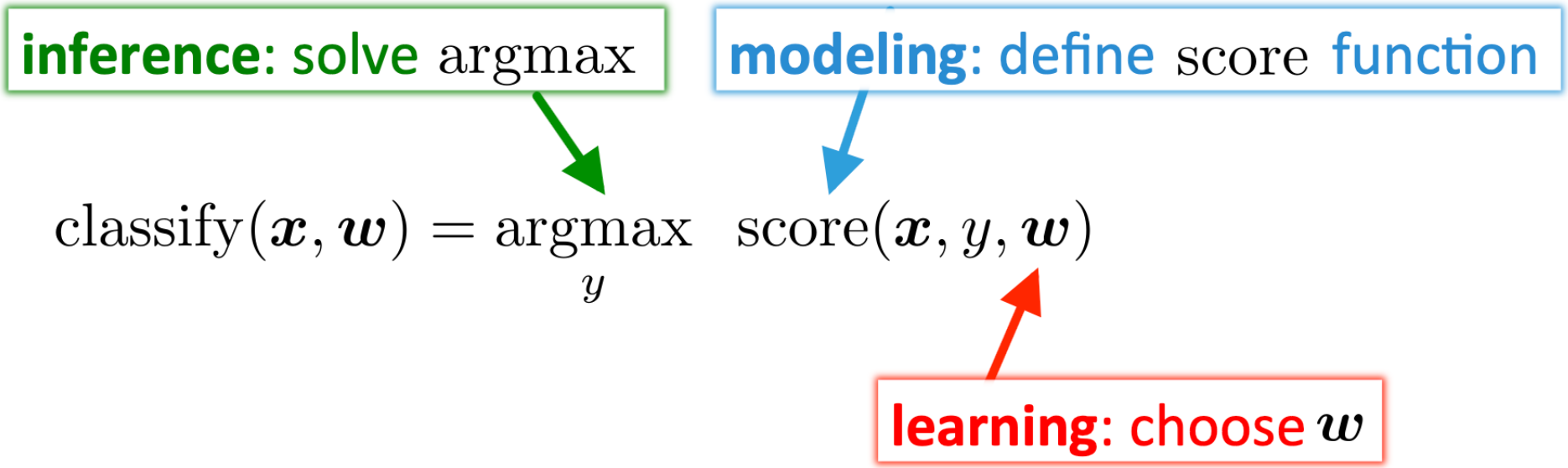
- **Modeling**: How do we assign a score to an (x,y) pair using parameters w?

Modeling, Inference, Learning



- **Inference**: How do we efficiently search over the space of all labels?

Modeling, Inference, Learning



- **Learning**: How do we choose the weights ***w***?

Binary Sentiment Classification

$$\text{classify}(\boldsymbol{x}, \boldsymbol{w}) = \underset{y \in \{\text{positive,negative}\}}{\text{argmax}} \text{score}(\boldsymbol{x}, y, \boldsymbol{w})$$

a sequence of words
(a sentence or document)

sentiment label

Binary Sentiment Classification

modeling: define score function

$$\text{classify}(\boldsymbol{x}, \boldsymbol{w}) = \underset{y \in \{\text{positive,negative}\}}{\text{argmax}} \; \text{score}(\boldsymbol{x}, y, \boldsymbol{w})$$

Linear Models

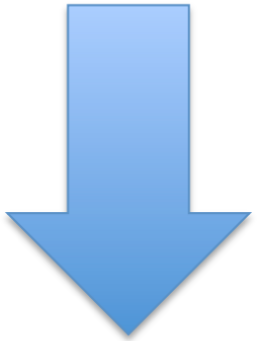
- Parameters are arranged in a vector **w**
- score function is linear in **w**:

$$\text{score}(\boldsymbol{x}, y, \mathbf{w}) = \mathbf{w}^\top \mathbf{f}(\boldsymbol{x}, y) = \sum_i w_i f_i(\boldsymbol{x}, y)$$

- **f**: vector of feature functions

Linear Models for Binary Sentiment Classification

$$\text{classify}(\boldsymbol{x}, \boldsymbol{w}) = \underset{y \in \{\text{positive,negative}\}}{\text{argmax}} \quad \text{score}(\boldsymbol{x}, y, \boldsymbol{w})$$



$$\text{classify}(\boldsymbol{x}, \mathbf{w}) = \underset{y \in \{\text{positive,negative}\}}{\text{argmax}} \quad \mathbf{w}^\top \mathbf{f}(\boldsymbol{x}, y)$$

- How do we define **f** ?

Features for NLP

- NLP datasets include inputs and outputs
- Features are usually not included
- You have to define your own features
- Contrast this with UCI datasets, which include a fixed-length **dense** feature vector for every instance
- In (traditional) NLP, features usually **sparse**

Defining Features

- This is a large part of NLP
- Last 30 years: feature engineering
- Last 10 years: representation learning

- Learning representations doesn't mean that we don't have to look at the data or the output!
- There's still plenty of engineering required in representation learning

N-gram

- **N-gram**: a sequence consisting of consecutive **n** words
- E.g., “Obama is going to”
  - unigram: “Obama”, “is”, “going”, “to”
  - bigram: “Obama is”, “is going”, “going to”
  - trigram: “Obama is going”, “is going to”

  …

Unigram Binary Features

- Two example features:

$$f_1(\boldsymbol{x}, y) = \mathbb{I}[y = \text{ positive}] \wedge \mathbb{I}[\boldsymbol{x} \text{ contains } \textit{great}]$$
$$f_2(\boldsymbol{x}, y) = \mathbb{I}[y = \text{ negative}] \wedge \mathbb{I}[\boldsymbol{x} \text{ contains } \textit{great}]$$

where $\mathbb{I}[S] = 1$ if S is true, 0 otherwise

- We usually think in terms of feature **templates**
- Unigram binary feature template:

$$f^{\text{u,b}}(\boldsymbol{x}, y) = \mathbb{I}[y = \text{ label}] \wedge \mathbb{I}[\boldsymbol{x} \text{ contains } \textit{word}]$$

- To create features, this feature template is instantiated for particular labels and words

Higher-Order Binary Feature Templates

- Unigram binary template:

$$f^{\mathrm{u,b}}(\boldsymbol{x}, y) = \mathbb{I}[y = \text{ label}] \wedge \mathbb{I}[\boldsymbol{x} \text{ contains } \textit{word}]$$

- Bigram binary template:

$$f^{\mathrm{b,b}}(\boldsymbol{x}, y) = \mathbb{I}[y = \text{ label}] \wedge \mathbb{I}[\boldsymbol{x} \text{ contains } \textit{“word1 word2”}]$$

- Trigram binary template:

…

Text Classification

- Datasets
- Classification
  - Modeling
  - **Inference**
  - Learning

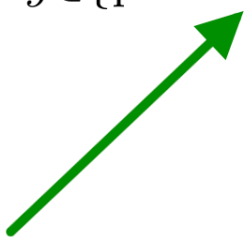
inference: solve argmax

$$\text{classify}(\boldsymbol{x}, \boldsymbol{w}) = \underset{y}{\text{argmax}} \;\; \text{score}(\boldsymbol{x}, y, \boldsymbol{w})$$

- **Inference**: How do we efficiently search over the space of all labels?

Inference for Text Classification

$$\text{classify}(\boldsymbol{x}, \boldsymbol{w}) = \underset{y \in \{\text{positive}, \text{negative}\}}{\text{argmax}} \text{score}(\boldsymbol{x}, y, \boldsymbol{w})$$

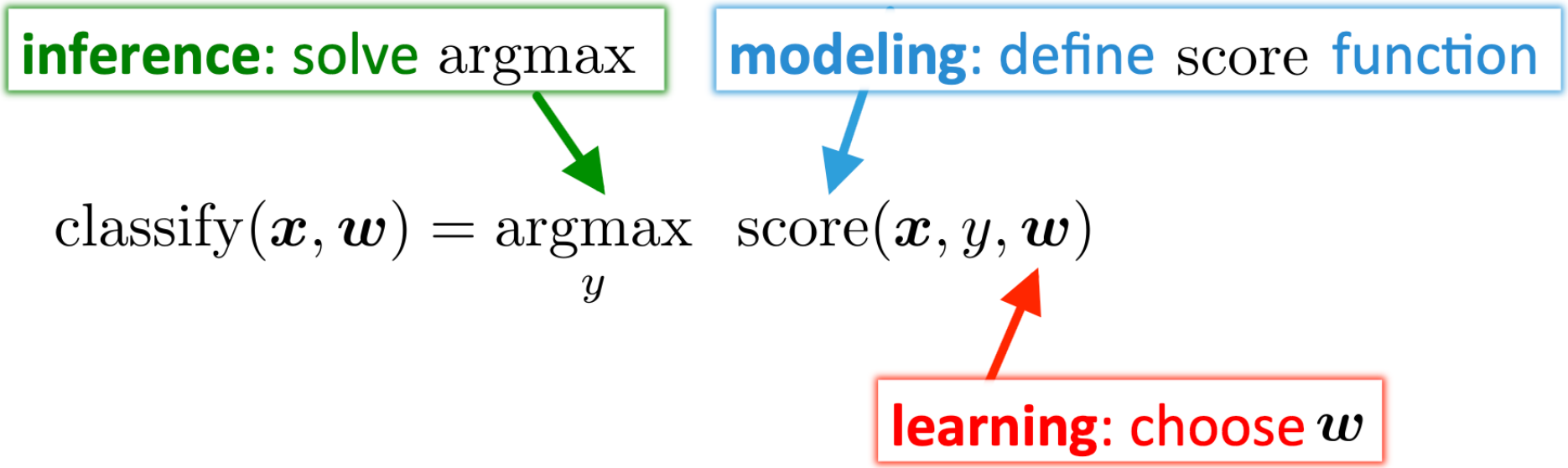


- trivial (loop over labels)

Text Classification

- Datasets
- Classification
  - Modeling
  - Inference
  - **Learning**

Modeling, Inference, Learning



- **Learning**: How should we choose values for the weights ***w***?

Text Classification

- Modeling
- Inference
- Learning
  - **empirical risk minimization**
  - surrogate loss functions
  - gradient-based optimization

Cost Functions

- Cost function: scores outputs against a gold standard

$$\text{cost} : \mathcal{L} \times \mathcal{L} \to \mathbb{R}_{\geq 0}$$

- Should be as close as possible to the actual evaluation metric for your task
- for classification, what cost should we use?

$$\text{cost}(y, y') = \mathbb{I}[y \neq y']$$

Risk Minimization

- given training data: $\mathcal{T} = \{\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle\}_{i=1}^{|\mathcal{T}|}$

  where each $y^{(i)} \in \mathcal{L}$ is a label

- assume data is drawn iid (independently and identically distributed) from (unknown) joint distribution $P(\boldsymbol{x}, y)$
- we want to solve the following:

$$\hat{\boldsymbol{w}} = \underset{\boldsymbol{w}}{\operatorname{argmin}} \; \mathbb{E}_{P(\boldsymbol{x},y)} \left[\operatorname{cost}(y, \operatorname{classify}(\boldsymbol{x}, \boldsymbol{w}))\right]$$

problem: *P* is unknown

Empirical Risk Minimization (Vapnik et al.)

- Replace expectation with sum over examples:

$$\hat{\boldsymbol{w}} = \underset{\boldsymbol{w}}{\operatorname{argmin}} \; \mathbb{E}_{P(\boldsymbol{x},y)} \left[\operatorname{cost}(y, \operatorname{classify}(\boldsymbol{x}, \boldsymbol{w})) \right]$$

$$\hat{\boldsymbol{w}} = \underset{\boldsymbol{w}}{\operatorname{argmin}} \; \sum_{i=1}^{|\mathcal{T}|} \operatorname{cost}(y^{(i)}, \operatorname{classify}(\boldsymbol{x}^{(i)}, \boldsymbol{w}))$$

problem: NP-hard even for binary classification with linear models

- Solution: replace "cost loss" (also called "0-1" loss) with a surrogate function that is easier to optimize

$$\hat{\boldsymbol{w}} = \underset{\boldsymbol{w}}{\operatorname{argmin}} \sum_{i=1}^{|\mathcal{T}|} \operatorname{cost}(y^{(i)}, \operatorname{classify}(\boldsymbol{x}^{(i)}, \boldsymbol{w}))$$

generalize to permit any loss function

$$\hat{\boldsymbol{w}} = \underset{\boldsymbol{w}}{\operatorname{argmin}} \sum_{i=1}^{|\mathcal{T}|} \operatorname{loss}(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{w})$$

- Cost loss / 0-1 loss: $\operatorname{loss}_{\text{cost}}(\boldsymbol{x}, y, \boldsymbol{w}) = \operatorname{cost}(y, \operatorname{classify}(\boldsymbol{x}, \boldsymbol{w}))$

Text Classification

- Modeling
- Inference
- Learning
  - empirical risk minimization
  - **surrogate loss functions**
  - gradient-based optimization

Surrogate Loss Functions

- Cost loss / 0-1 loss: $\text{loss}_{\text{cost}}(\boldsymbol{x}, y, \boldsymbol{w}) = \text{cost}(y, \text{classify}(\boldsymbol{x}, \boldsymbol{w}))$
  - why is this so difficult to optimize? not necessarily continuous, can't use gradient-based optimization
- max-score loss: $\text{loss}_{\text{maxscore}}(\boldsymbol{x}, y, \boldsymbol{w}) = -\text{score}(\boldsymbol{x}, y, \boldsymbol{w})$
- perceptron loss: $\text{loss}_{\text{perc}}(\boldsymbol{x}, y, \boldsymbol{w}) = -\text{score}(\boldsymbol{x}, y, \boldsymbol{w}) + \max_{y' \in \mathcal{L}} \text{score}(\boldsymbol{x}, y', \boldsymbol{w})$
  - loss function underlying perceptron algorithm (Rosenblatt, 1957-58)

Surrogate Loss Functions

- Cost loss / 0-1 loss: $\text{loss}_{\text{cost}}(\boldsymbol{x}, y, \boldsymbol{w}) = \text{cost}(y, \text{classify}(\boldsymbol{x}, \boldsymbol{w}))$
- perceptron loss: $\text{loss}_{\text{perc}}(\boldsymbol{x}, y, \boldsymbol{w}) = -\text{score}(\boldsymbol{x}, y, \boldsymbol{w}) + \max_{y' \in \mathcal{L}} \text{score}(\boldsymbol{x}, y', \boldsymbol{w})$
- hinge loss:

$$\text{loss}_{\text{hinge}}(\boldsymbol{x}, y, \boldsymbol{w}) = -\text{score}(\boldsymbol{x}, y, \boldsymbol{w}) + \max_{y' \in \mathcal{L}} (\text{score}(\boldsymbol{x}, y', \boldsymbol{w}) + \text{cost}(y, y'))$$

  - loss function underlying support vector machines

Log Loss

$$\text{loss}_{\log}(\boldsymbol{x}, y, \boldsymbol{w}) = -\log p_{\boldsymbol{w}}(y \mid \boldsymbol{x})$$

- Minimize negative log of conditional probability of output given input
  - sometimes called “maximizing conditional likelihood”
- But we don’t have a probabilistic model, we just have a score function

Score → Probability

- Can turn score into probability by exponentiating (to make it positive) and normalizing:

$$p_{\boldsymbol{w}}(y \mid \boldsymbol{x}) \propto \exp\{\text{score}(\boldsymbol{x}, y, \boldsymbol{w})\}$$

$$p_{\boldsymbol{w}}(y \mid \boldsymbol{x}) = \frac{\exp\{\text{score}(\boldsymbol{x}, y, \boldsymbol{w})\}}{\sum_{y' \in \mathcal{L}} \exp\{\text{score}(\boldsymbol{x}, y', \boldsymbol{w})\}}$$

- This is open called a “softmax” function

Log Loss

$$\begin{aligned}
\text{loss}_{\text{log}}(\boldsymbol{x}, y, \boldsymbol{w}) &= -\log p_{\boldsymbol{w}}(y \mid \boldsymbol{x}) \\
&= -\log \frac{\exp\{\text{score}(\boldsymbol{x}, y, \boldsymbol{w})\}}{\sum_{y' \in \mathcal{L}} \exp\{\text{score}(\boldsymbol{x}, y', \boldsymbol{w})\}} \\
&= -\text{score}(\boldsymbol{x}, y, \boldsymbol{w}) + \log \sum_{y' \in \mathcal{L}} \exp\{\text{score}(\boldsymbol{x}, y', \boldsymbol{w})\}
\end{aligned}$$

- Similar to perceptron loss!
- Replace max with "softmax" (a different kind of softmax)

$$\text{loss}_{\text{perc}}(\boldsymbol{x}, y, \boldsymbol{w}) = -\text{score}(\boldsymbol{x}, y, \boldsymbol{w}) + \max_{y' \in \mathcal{L}} \text{score}(\boldsymbol{x}, y', \boldsymbol{w})$$

Log Loss

$$\begin{aligned}
\text{loss}_{\log}(\boldsymbol{x}, y, \boldsymbol{w}) &= -\log p_{\boldsymbol{w}}(y \mid \boldsymbol{x}) \\
&= -\log \frac{\exp\{\text{score}(\boldsymbol{x}, y, \boldsymbol{w})\}}{\sum_{y' \in \mathcal{L}} \exp\{\text{score}(\boldsymbol{x}, y', \boldsymbol{w})\}} \\
&= -\text{score}(\boldsymbol{x}, y, \boldsymbol{w}) + \log \sum_{y' \in \mathcal{L}} \exp\{\text{score}(\boldsymbol{x}, y', \boldsymbol{w})\}
\end{aligned}$$

- Log loss is used in:
  - logistic regression classifiers,
  - conditional random fields,
  - maximum entropy (“maxent”) models

Regularized Empirical Risk Minimization

$$\hat{\boldsymbol{w}} = \operatorname*{argmin}_{\boldsymbol{w}} \; \mathbb{E}_{P(\boldsymbol{x},y)} \left[\text{cost}(y, \text{classify}(\boldsymbol{x}, \boldsymbol{w})) \right]$$

$$\hat{\boldsymbol{w}} = \operatorname*{argmin}_{\boldsymbol{w}} \; \sum_{i=1}^{|\mathcal{T}|} \text{loss}(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{w}) + \lambda R(\boldsymbol{w})$$

regularization strength

regularization term

Regularization Terms

$$\hat{\boldsymbol{w}} = \underset{\boldsymbol{w}}{\operatorname{argmin}} \sum_{i=1}^{|\mathcal{T}|} \operatorname{loss}(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{w}) + \lambda R(\boldsymbol{w})$$

- Most common: penalize large parameter values
- Intuition: large parameters might be instances of overfitting
- Examples:
  - L_2 **regularization**: (also called Tikhonov regularization or ridge regression)
  - L_1 **regularization**: (also called basis pursuit or LASSO)

Regularization Terms

- **L_2 regularization**: $R_{\mathrm{L2}}(\mathbf{w}) = ||\mathbf{w}||_2^2 = \sum_i w_i^2$
  - differentiable, widely-used
- **L_1 regularization:** $R_{\mathrm{L1}}(\mathbf{w}) = ||\mathbf{w}||_1 = \sum_i |w_i|$
  - not differentiable (but is subdifferentiable)
  - leads to sparse solutions (many parameters become zero!)

Text Classification

- Modeling
- Inference
- Learning
  - empirical risk minimization
  - surrogate loss functions
  - **gradient-based optimization**

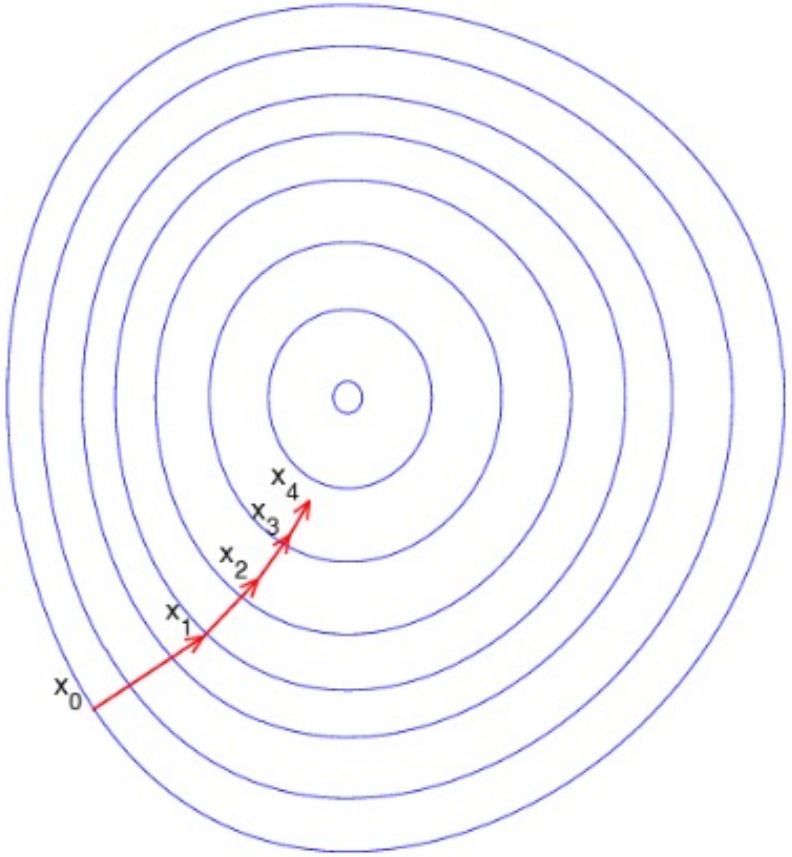
Gradient Descent

- Minimizes a function *F* by taking steps in proportion to the negative of the gradient:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta^{(t)} \nabla F(\boldsymbol{\theta}^{(t)})$$

$\eta^{(t)}$: stepsize at iteration t
$\nabla F(\boldsymbol{\theta}^{(t)})$: gradient of objective function

- With conditions on stepsize and objective function, will converge to local minimum



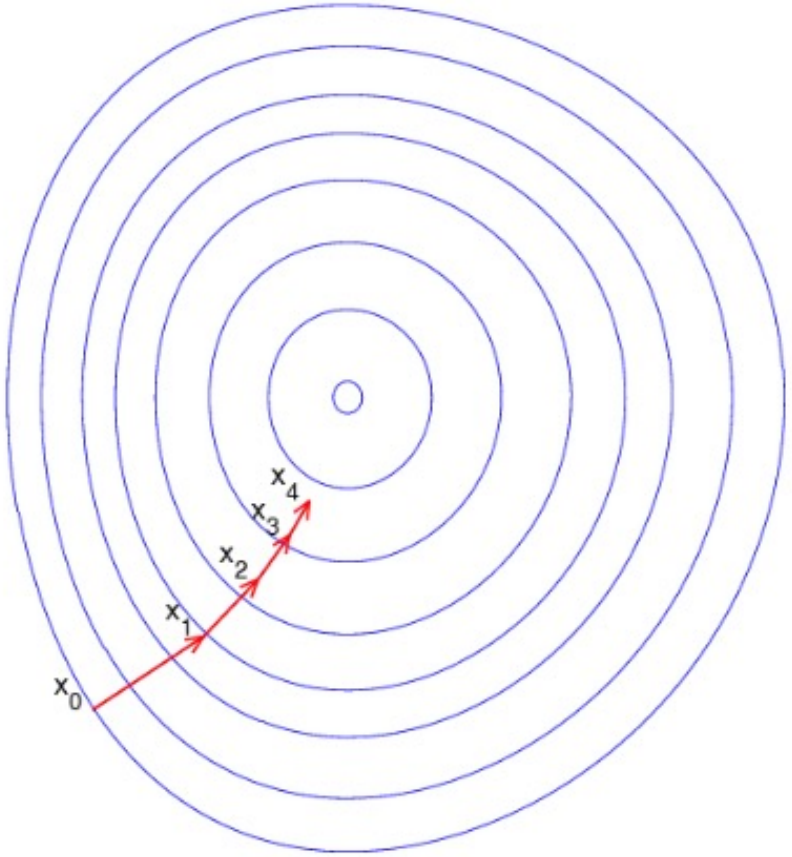
Gradient Descent

- Minimizes a function *F* by taking steps in proportion to the negative of the gradient:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta^{(t)} \nabla F(\boldsymbol{\theta}^{(t)})$$

$\eta^{(t)}$: stepsize at iteration t
$\nabla F(\boldsymbol{\theta}^{(t)})$: gradient of objective function

- Efficiency concern: F is a sum over all training examples! every parameter update requires iterating through entire training set. "**batch" algorithm**



Batch Gradient Descent

- Batch Gradient Descent Algorithm

Input: data $\{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}$ $((1 \leq i \leq N)$, learning rate α_t.

(I) $t = 0$; Make an initial guess θ_0;

(II) Iterate until termination condition is met

(i) $\theta_{t+1} = \theta_t - \alpha_t \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \ell(f(\mathbf{x}^{(i)}; \theta_t), \mathbf{y}^{(i)})$;

(ii) $t = t + 1$;

Stochastic Gradient Descent

- Stochastic Gradient Descent Algorithm

Input: data $\{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}$ $((1 \leq i \leq N)$, learning rate α_t.

(I) $t = 0$; Make an initial guess θ_0;

(II) Iterate until termination condition is met

(i) Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set

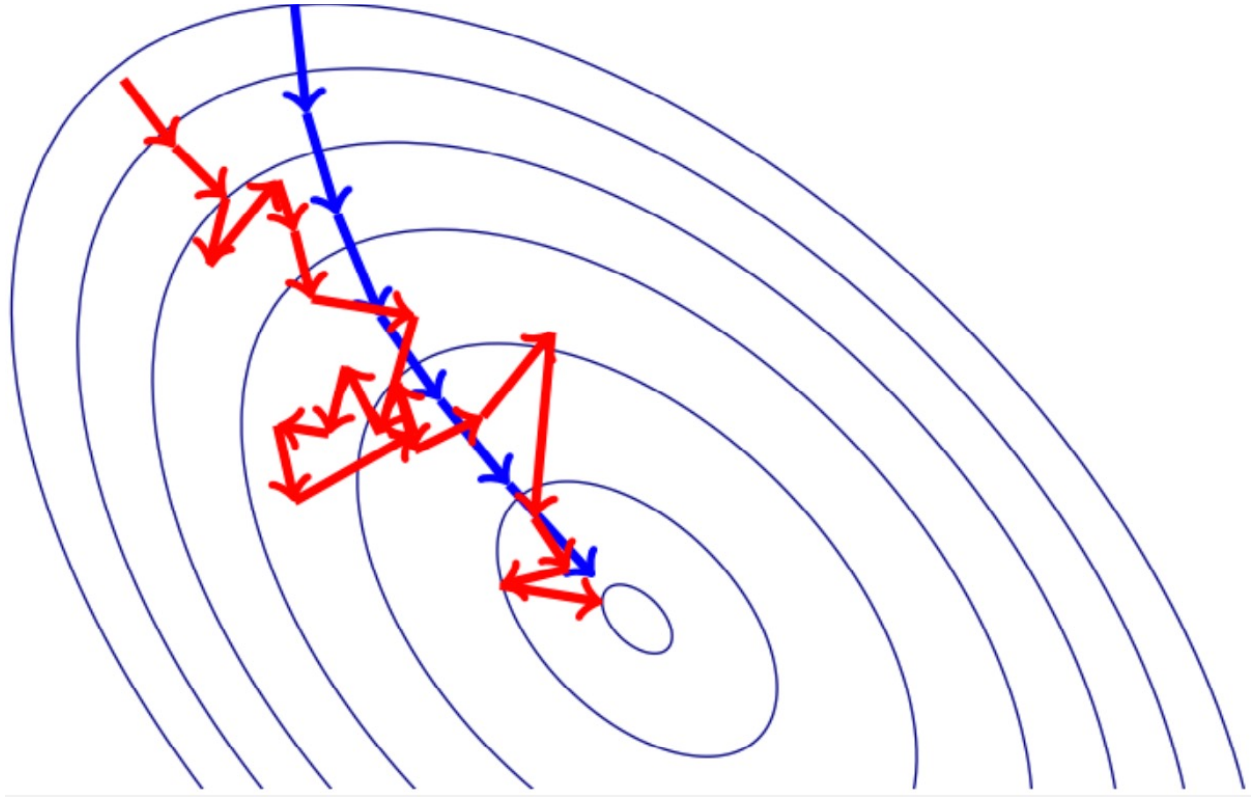
(ii) $\theta_{t+1} = \theta_t - \alpha_t \sum_{i=1}^{N} \nabla_\theta \ell(f(\mathbf{x}^{(i)}; \theta_t), \mathbf{y}^{(i)})$;

(iii) $t = t + 1$;

Stochastic Gradient Descent

- Applicable when objective function is a sum
- Like gradient descent, except calculates gradient on a single example at a time ("online") or on a small set of examples ("mini-batch")
- Converges much faster than (batch) gradient descent
- With conditions on stepsize and objective function, will converge to local minimum
- There are many popular variants:
  - SGD+momentum, AdaGrad, AdaDelta, Adam, RMSprop, etc.

Stochastic Gradient Descent



What if F is not differentiable?

- Some loss functions are not differentiable:

$$\text{loss}_{\text{perc}}(\boldsymbol{x}, y, \boldsymbol{w}) = -\text{score}(\boldsymbol{x}, y, \boldsymbol{w}) + \max_{y' \in \mathcal{L}} \text{score}(\boldsymbol{x}, y', \boldsymbol{w})$$

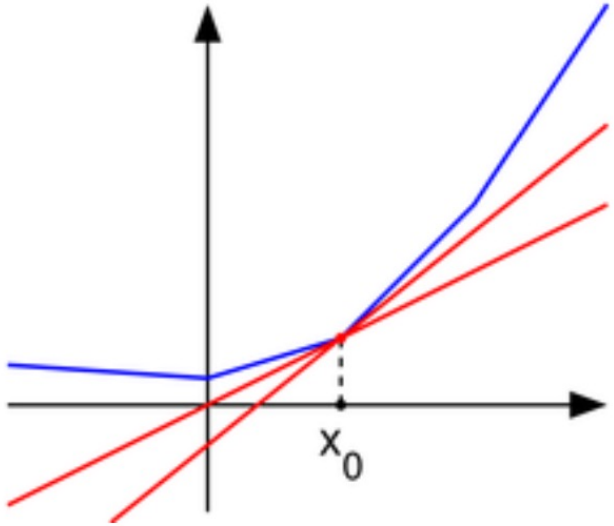
$$\text{loss}_{\text{hinge}}(\boldsymbol{x}, y, \boldsymbol{w}) = -\text{score}(\boldsymbol{x}, y, \boldsymbol{w}) + \max_{y' \in \mathcal{L}} (\text{score}(\boldsymbol{x}, y', \boldsymbol{w}) + \text{cost}(y, y'))$$

- But they *are* **subdifferentiable**, so we can compute **subgradients** and use (stochastic) subgradient method (not descent!)
  - So we need to keep track of the best solution so far

Subderivatives

- **Subderivative**: generalization of derivative for nondifferentiable, convex functions

- There may be multiple subderivatives at a point (red lines)



- This set is called the **subdifferential**

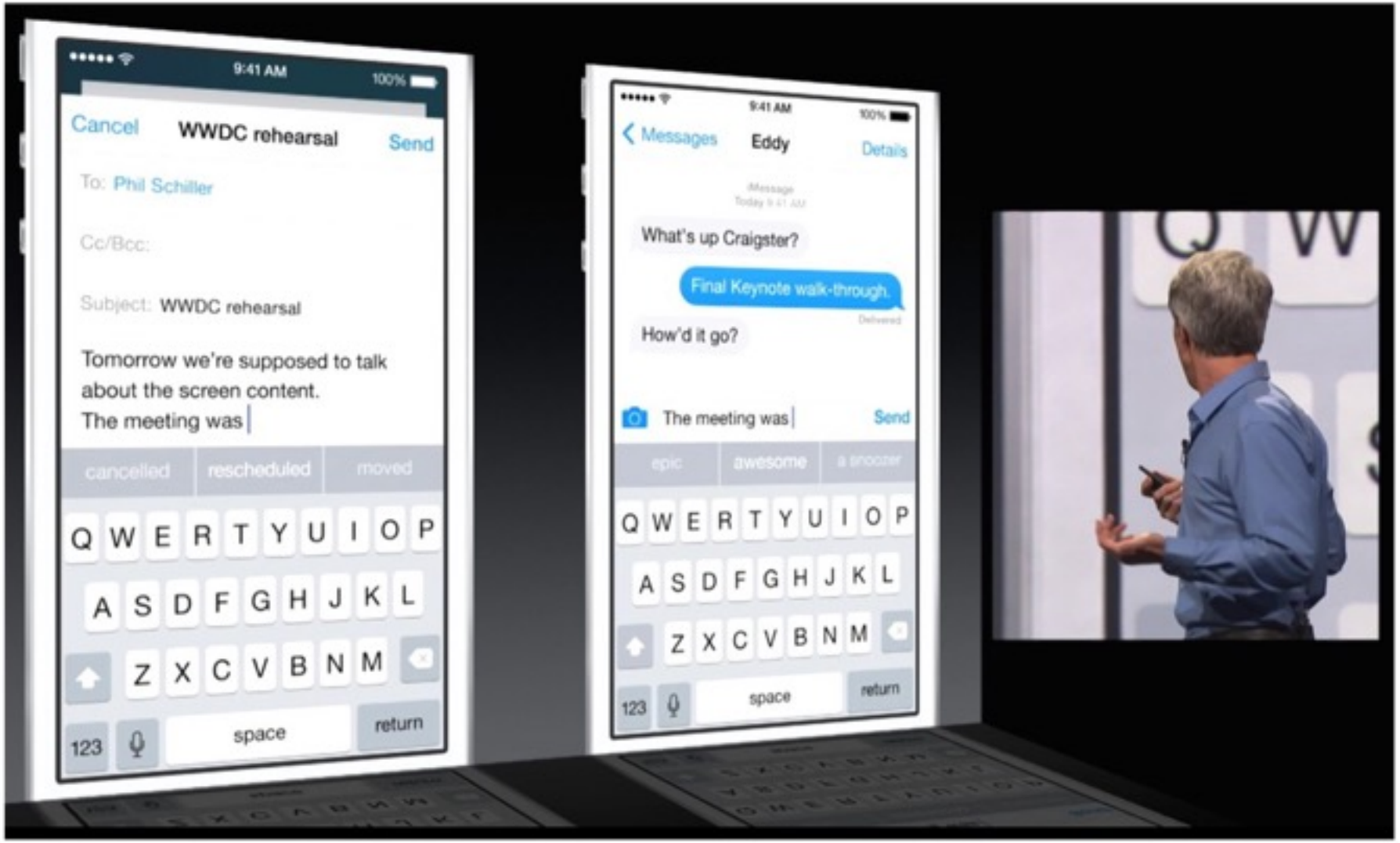
- A convex function *g* is differentiable at point x_0 if and only if the subdifferential of *g* at x_0 contains only the derivative of *g* at x_0

Language Modeling

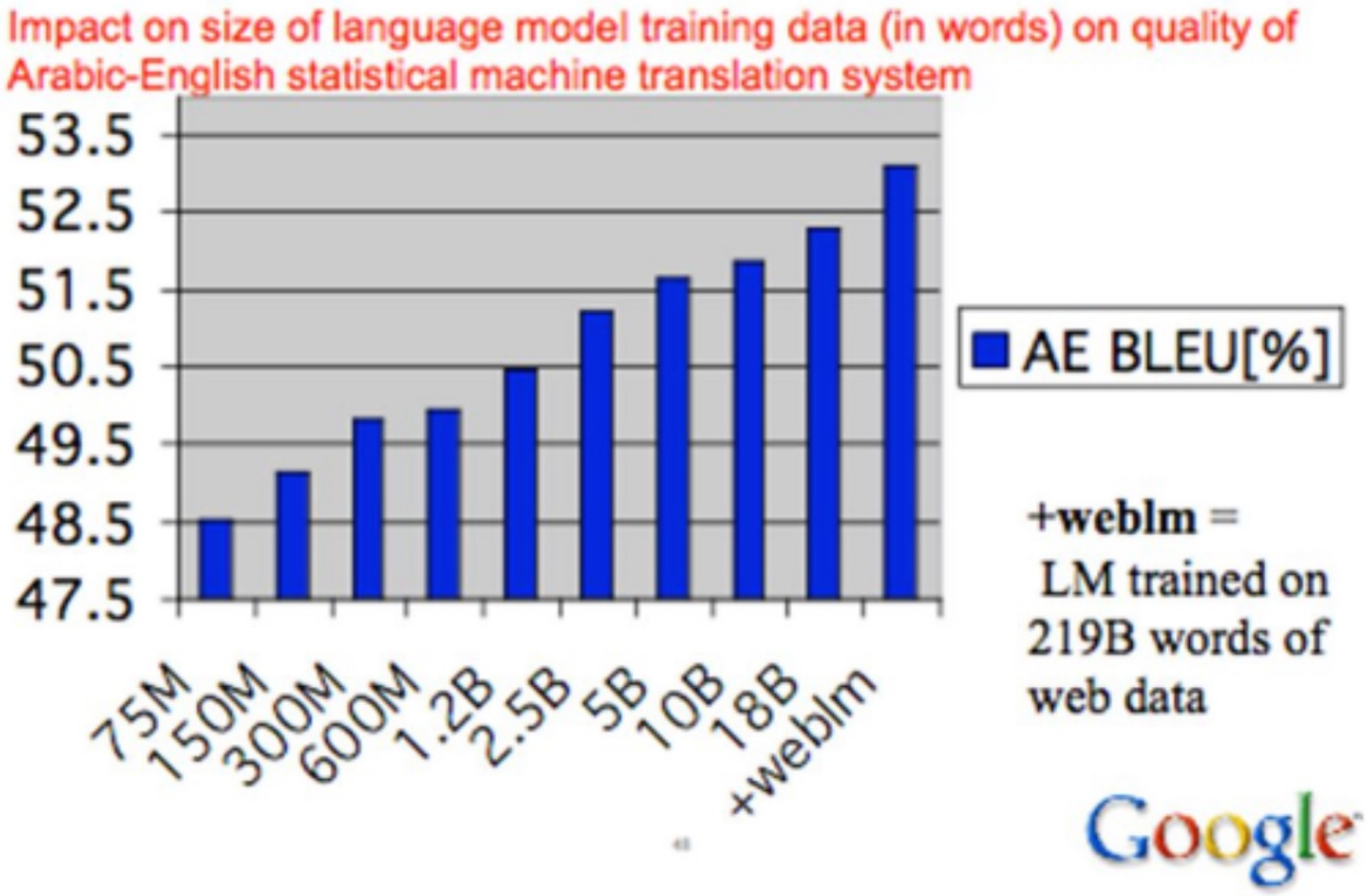
Probabilistic Language Models

- Language modeling: assign probabilities to sentences
- Why?
  - machine translation:
    - P(**high** winds tonite) > P(**large** winds tonite)
  - spelling correction:
    - The office is about fifteen **minuets** from my house
  - speech recognition:
    - P(I saw a van) >> P(eyes awe of an)
  - summarization, question answering, etc.!

1


9:41 AM
100%
Cancel
WWDC rehearsal
Send
To: Phil Schiller
Cc/Bcc:
Subject: WWDC rehearsal
Tomorrow we're supposed to talk about the screen content.
The meeting was
cancelled
rescheduled
moved
Q W E R T Y U I O P
A S D F G H J K L
Z X C V B N M
123
space
return
9:41 AM
100%
Messages
Eddy
Details
What's up Craigster?
Final Keynote walk-through.
Delivered
How'd it go?
The meeting was
Send
epic
awesome
Q W E R T Y U I O P
A S D F G H J K L
Z X C V B N M
123
space
return

Language Modeling for Machine Translation



Probabilistic Language Modeling

- Goal: compute the probability of a sequence of words:

$$P(\boldsymbol{w}) = P(w_1, w_2, ..., w_n)$$

- Related task: probability of next word:

$$P(w_4 \mid w_1, w_2, w_3)$$

- A model that computes either of these:

$$P(\boldsymbol{w}) \quad \text{or} \quad P(w_k \mid w_1, w_2, ..., w_{k-1})$$

is called a **language model (LM)**

How to compute $P(\boldsymbol{w})$

- How to compute this joint probability:
  - P(its, water, is, so, transparent, that)
- Intuition: let's rely on the Chain Rule of Probability

Reminder: Chain Rule

- Factor joint probability into product of conditional probabilities:

$$P(w_1, w_2, ..., w_n) = P(w_1)P(w_2 \mid w_1)P(w_3 \mid w_1, w_2) \ldots P(w_n \mid w_1, w_2, ..., w_{n-1})$$

- We have not yet made any independence assumptions

Chain Rule for computing joint probability of words in sentence

$$P(w_1, w_2, ..., w_n) = \prod_i P(w_i \mid w_1, w_2, ..., w_{i-1})$$

P(“*its water is so transparent*”) =

P(*its*) × P(*water* | *its*) × P(*is* | *its water*)

× P(*so* | *its water is*) × P(*transparent* | *its water is so*)

How to estimate these probabilities

- Could we just count and divide?

$$P(\text{the} \mid \text{its water is so transparent that}) = \frac{Count(\text{its water is so transparent that the})}{Count(\text{its water is so transparent that})}$$

- No! too many possible sentences!

- We'll never see enough data for estimating these

Markov Assumption

- Simplifying assumption:

$$P(\text{the} \mid \text{its water is so transparent that}) \approx P(\text{the} \mid \text{that})$$

- Or maybe:

$$P(\text{the} \mid \text{its water is so transparent that}) \approx P(\text{the} \mid \text{transparent that})$$

- I.e., we approximate each component in the product:

$$P(w_i \mid w_1, ..., w_{i-2}, w_{i-1}) \approx P(w_i \mid w_{i-k}, ..., w_{i-2}, w_{i-1})$$

Simplest case: Unigram model

$$P(w_1, w_2, ..., w_n) = \prod_i P(w_i)$$

- Automatically generated sentences from a unigram model:

```
fifth an of futures the an incorporated a a the
inflation most dollars quarter in is mass

thrift did eighty said hard 'm july bullish

that or limited the
```

Bigram model

- condition on the previous word:

$$P(w_1, w_2, \ldots, w_n) = \prod_i P(w_i \mid w_{i-1})$$

- Automatically generated sentences from a unigram model:

```
texaco rose one in this issue is pursuing growth in a boiler
house said mr. gurria mexico 's motion control proposal
without permission from five hundred fifty five yen

outside new car parking lot of the agreement reached

this would be a record november
```

n-gram models

- We can extend to trigrams, 4-grams, 5-grams
- In general this is an insufficient model of language
  - because language has **long-distance dependencies**:

    "The computer which I had just put into the machine room on the fi6h floor crashed."

- But we can often get away with n-gram models

Estimating bigram probabilities

- The maximum likelihood estimate (MLE)

$$P(w_i \mid w_{i-1}) = \frac{\text{count}(w_{i-1}, w_i)}{\text{count}(w_{i-1})}$$

An example

$$P(w_i \mid w_{i-1}) = \frac{\text{count}(w_{i-1}, w_i)}{\text{count}(w_{i-1})}$$

<s> I am Sam </s>
<s> Sam I am </s>
<s> I do not like green eggs and ham </s>

$P(\texttt{I} \mid \texttt{<s>}) = \frac{2}{3} = .67$ $P(\texttt{Sam} \mid \texttt{<s>}) = \frac{1}{3} = .33$ $P(\texttt{am} \mid \texttt{I}) = \frac{2}{3} = .67$

$P(\texttt{</s>} \mid \texttt{Sam}) = \frac{1}{2} = 0.5$ $P(\texttt{Sam} \mid \texttt{am}) = \frac{1}{2} = .5$ $P(\texttt{do} \mid \texttt{I}) = \frac{1}{3} = .33$

More examples:
Berkeley Restaurant Project sentences

- Can you tell me about any good Cantonese restaurants close by
- Mid priced thai food is what i'm looking for
- Can you give me a listing of the kinds of food that are available

Raw bigram counts

- Counts from 9,222 sentences
- E.g., *"i want"* occurs 827 times

	i	want	to	eat	chinese	food	lunch	spend
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

Raw bigram probabilities

- normalize by unigram counts:

i	want	to	eat	chinese	food	lunch	spend
2533	927	2417	746	158	1093	341	278

- bigram probabilities

	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.33	0	0.0036	0	0	0	0.00079
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

J&M/SLP3

Bigram estimates of sentence probabilities

$$
\begin{aligned}
&P(\langle s\rangle\ I\ want\ english\ food\ \langle /s\rangle) = \\
&\quad P(I \mid \langle s\rangle) \\
&\quad \times\ P(want \mid I) \\
&\quad \times\ P(english \mid want) \\
&\quad \times\ P(food \mid english) \\
&\quad \times\ P(\langle /s\rangle \mid food) \\
&\qquad =\ .000031
\end{aligned}
$$

Practical Issues

- we do everything in log space
  - avoid underflow
  - also adding is faster than multiplying

$$\log(p_1 \times p_2 \times p_3) = \log p_1 + \log p_2 + \log p_3$$

Language Modeling Toolkits

- SRILM
  - http://www.speech.sri.com/projects/srilm/
- KenLM
  - https://kheafield.com/code/kenlm/

Evaluation: How good is our model?

- Does our language model prefer good sentences to bad ones?
  - assign higher probability to “real” or “frequently observed” sentences
    - than “ungrammatical” or “rarely observed” sentences?

Extrinsic evaluation of N-gram models

- Best evaluation for comparing models A and B
  - put each model in a task
    - spelling corrector, speech recognizer, MT system
  - run the task, get an accuracy for A and for B
    - how many misspelled words corrected properly
    - how many words translated correctly
  - compare accuracy for A and B

Intuition of Perplexity

- The Shannon Game:
  - how well can we predict the next word?

I always order pizza with cheese and ____

The 33rd President of the US was ____

I saw a ____

mushrooms 0.1

pepperoni 0.1

anchovies 0.01

...

fried rice 0.0001

...

and 1e-100

  - unigrams are terrible at this game (why?)
- A better model of a text is one which assigns a higher probability to the word that actually occurs

Probability of Held-out Data

- probability of held-out sentences:

$$\prod_i P(\boldsymbol{w}^{(i)})$$

- let's work with log-probabilities:

$$\log_2 \prod_i P(\boldsymbol{w}^{(i)}) = \sum_i \log_2 P(\boldsymbol{w}^{(i)})$$

- divide by number of words *M* in held-out sentences:

$$\frac{1}{M} \sum_i \log_2 P(\boldsymbol{w}^{(i)})$$

Probability -> Perplexity

- average log-probability of held-out words:

$$\ell = \frac{1}{M} \sum_i \log_2 P(\boldsymbol{w}^{(i)})$$

- perplexity:

$$\mathrm{PP} = 2^{-\ell}$$

Perplexity as branching factor

- Given a sentence consisting of random digits
- Perplexity of this sentence under a model that gives probability 1/10 to each digit?

$$\ell = \frac{1}{M} \log_2 P(w_1, w_2, ..., w_M)$$
$$= \frac{1}{M} \log_2 \prod_{i=1}^{M} \frac{1}{10}$$
$$= \frac{1}{M} \log_2 \left(\frac{1}{10}\right)^M$$
$$= \frac{1}{M} M \log_2 \frac{1}{10}$$

$$\mathrm{PP} = 2^{-\ell} = 10$$

Lower perplexity = better model

- Train: 38 million words
- Test: 1.5 million words

n-gram order:	unigram	bigram	trigram
perplexity:	962	170	109

Approximating Shakespeare

1 gram	–To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have –Hill he late speaks; or! a more to leg less first you enter
2 gram	–Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow. –What means, sir. I confess she? then all sorts, he is trim, captain.
3 gram	–Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done. –This shall forbid it should be branded, if renown made it empty.
4 gram	–King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in; –It cannot be but so.

Journal of Machine Learning Research 3 (2003) 1137–1155 Submitted 4/02; Published 2/03

A Neural Probabilistic Language Model

Yoshua Bengio BENGIOY@IRO.UMONTREAL.CA
Réjean Ducharme DUCHARME@IRO.UMONTREAL.CA
Pascal Vincent VINCENTP@IRO.UMONTREAL.CA
Christian Jauvin JAUVINC@IRO.UMONTREAL.CA
Département d'Informatique et Recherche Opérationnelle
Centre de Recherche Mathématiques
Université de Montréal, Montréal, Québec, Canada

- Idea: use a neural network for n-gram language modeling:

$$P_{\boldsymbol{\theta}}(w_t \mid w_{t-n+1}, \ldots, w_{t-2}, w_{t-1})$$

Journal of Machine Learning Research 3 (2003) 1137–1155 Submitted 4/02; Published 2/03

A Neural Probabilistic Language Model

Yoshua Bengio BENGIOY@IRO.UMONTREAL.CA
Réjean Ducharme DUCHARME@IRO.UMONTREAL.CA
Pascal Vincent VINCENTP@IRO.UMONTREAL.CA
Christian Jauvin JAUVINC@IRO.UMONTREAL.CA
Département d'Informatique et Recherche Opérationnelle
Centre de Recherche Mathématiques
Université de Montréal, Montréal, Québec, Canada

- This is not the earliest paper on using neural networks for n-gram language models, but it's the most well-known and first to scale up
- See paper for citations of earlier work

Neural Probabilistic Language Models (Bengio et al., 2003)

1.1 Fighting the Curse of Dimensionality with Distributed Representations

In a nutshell, the idea of the proposed approach can be summarized as follows:

1. associate with each word in the vocabulary a distributed *word feature vector* (a real-valued vector in $\mathbb{R}^m$),
2. express the joint *probability function* of word sequences in terms of the feature vectors of these words in the sequence, and
3. learn simultaneously the *word feature vectors* and the parameters of that *probability function*.

What is a neural network?

- Just think of a neural network as a function
- It has inputs and outputs
- “neural” typically means one type of functional building block (“neural layers”), but the term has broadened
- Neural modeling is now better thought of as a modeling strategy (leveraging “distributed representations” or “representation learning”), or a family of related methods

Notation

$\mathbf{u} =$ a vector

$u_i =$ entry *i* in the vector

$\mathbf{W} =$ a matrix

$w_{ij} =$ entry (*i*,*j*) in the matrix

$\boldsymbol{x} =$ a structured object

$x_i =$ entry *i* in the structured object

neural layer = affine transform + nonlinearity

$$\mathbf{z}^{(1)} = g\left(\underbrace{\mathbf{U}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}}_{\text{affine transform}}\right)$$

nonlinearity

- This is a single “layer” of a neural network
- Input vector is $\mathbf{x}$
- $\mathbf{U}^{(0)}$ and $\mathbf{b}^{(0)}$ are parameters

neural layer = affine transform + nonlinearity

$$\mathbf{z}^{(1)} = g\left(\mathbf{U}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$

"hidden units"

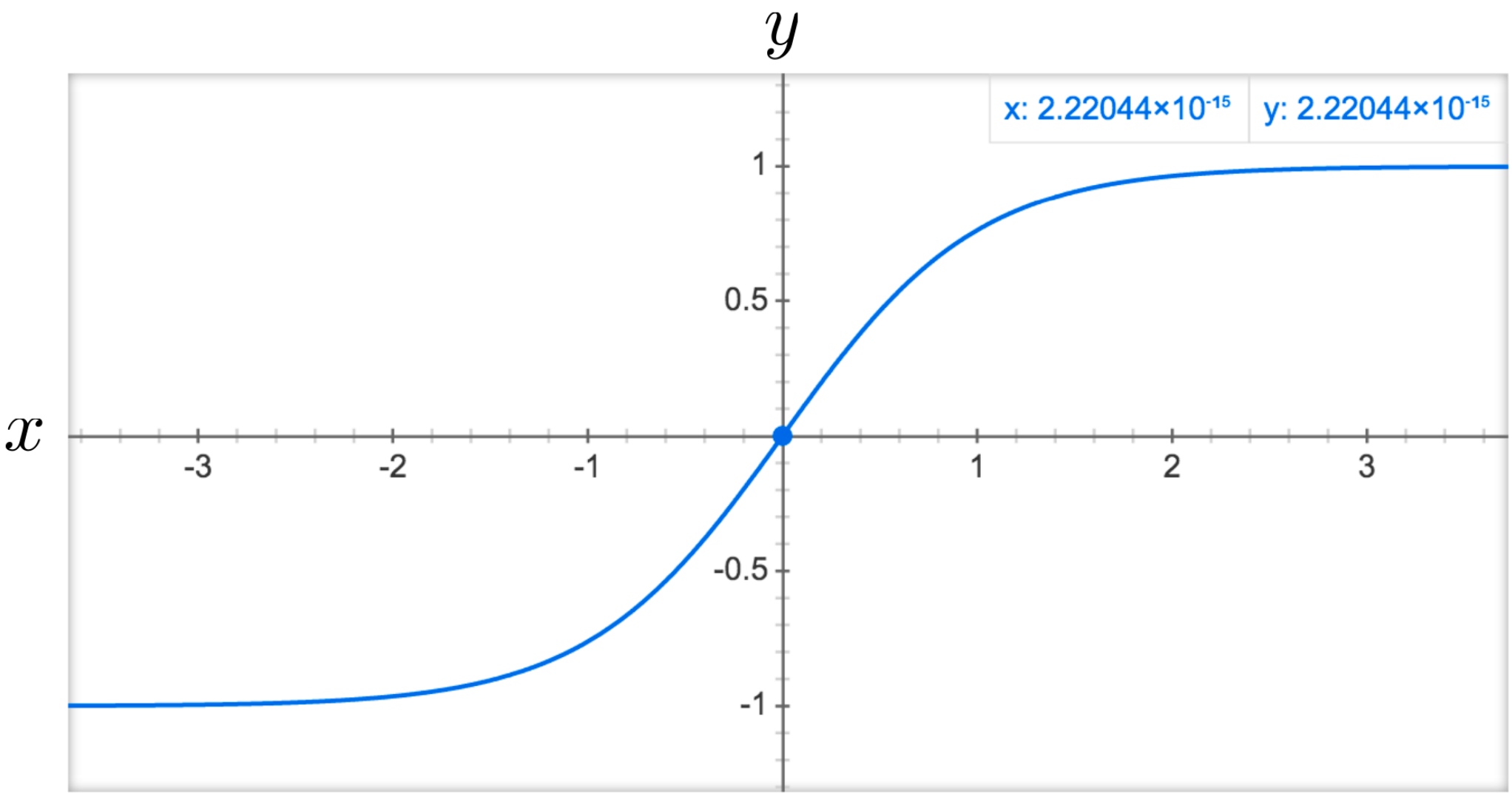
- Vector of "hidden units" is $\mathbf{z}^{(1)}$
- Think of these as features computed from $\mathbf{x}$

Nonlinearities

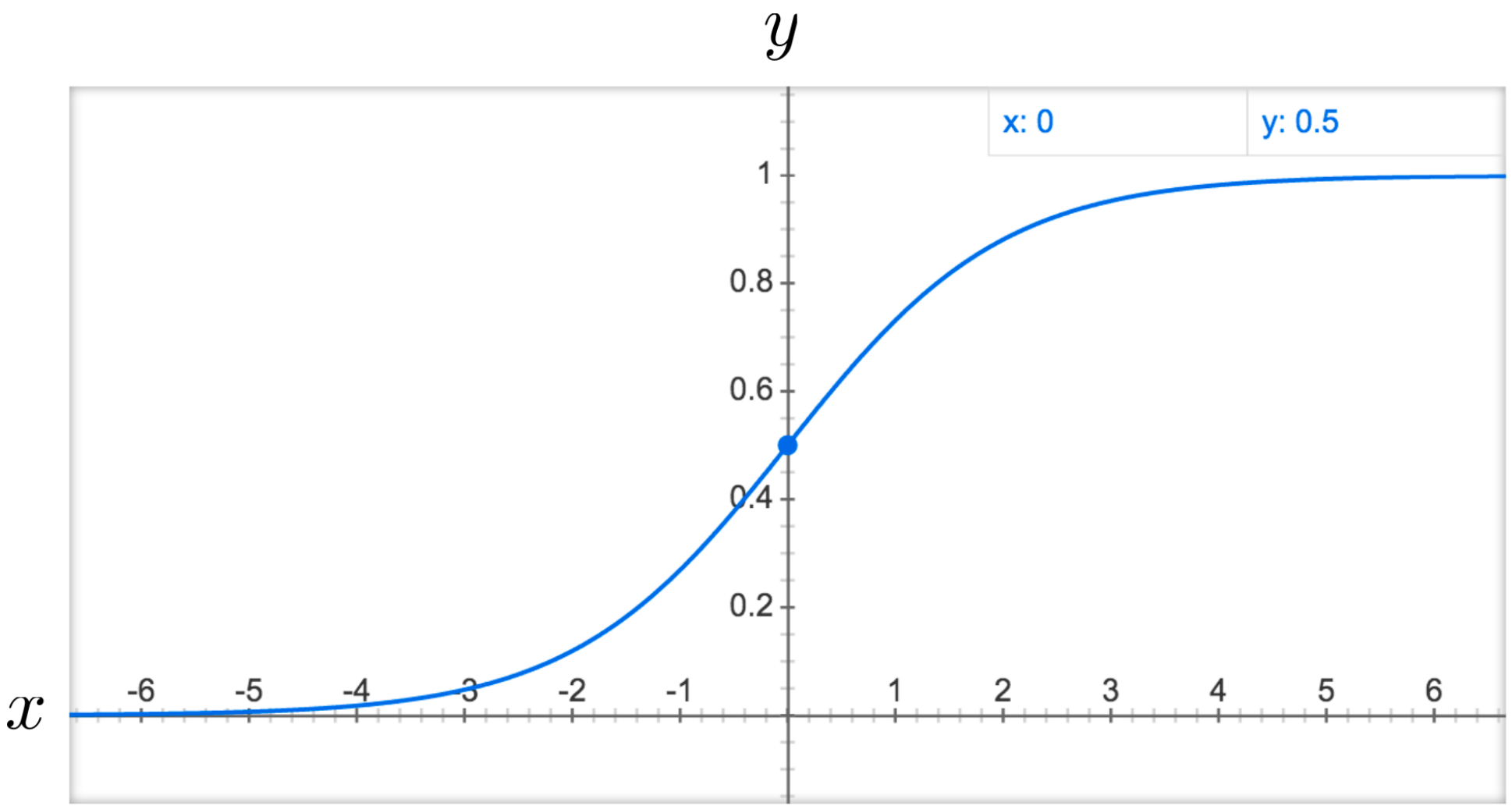
$$\mathbf{z}^{(1)} = g\left(\mathbf{U}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$

- Most common: elementwise application of *g* function to each entry in vector
- Examples...

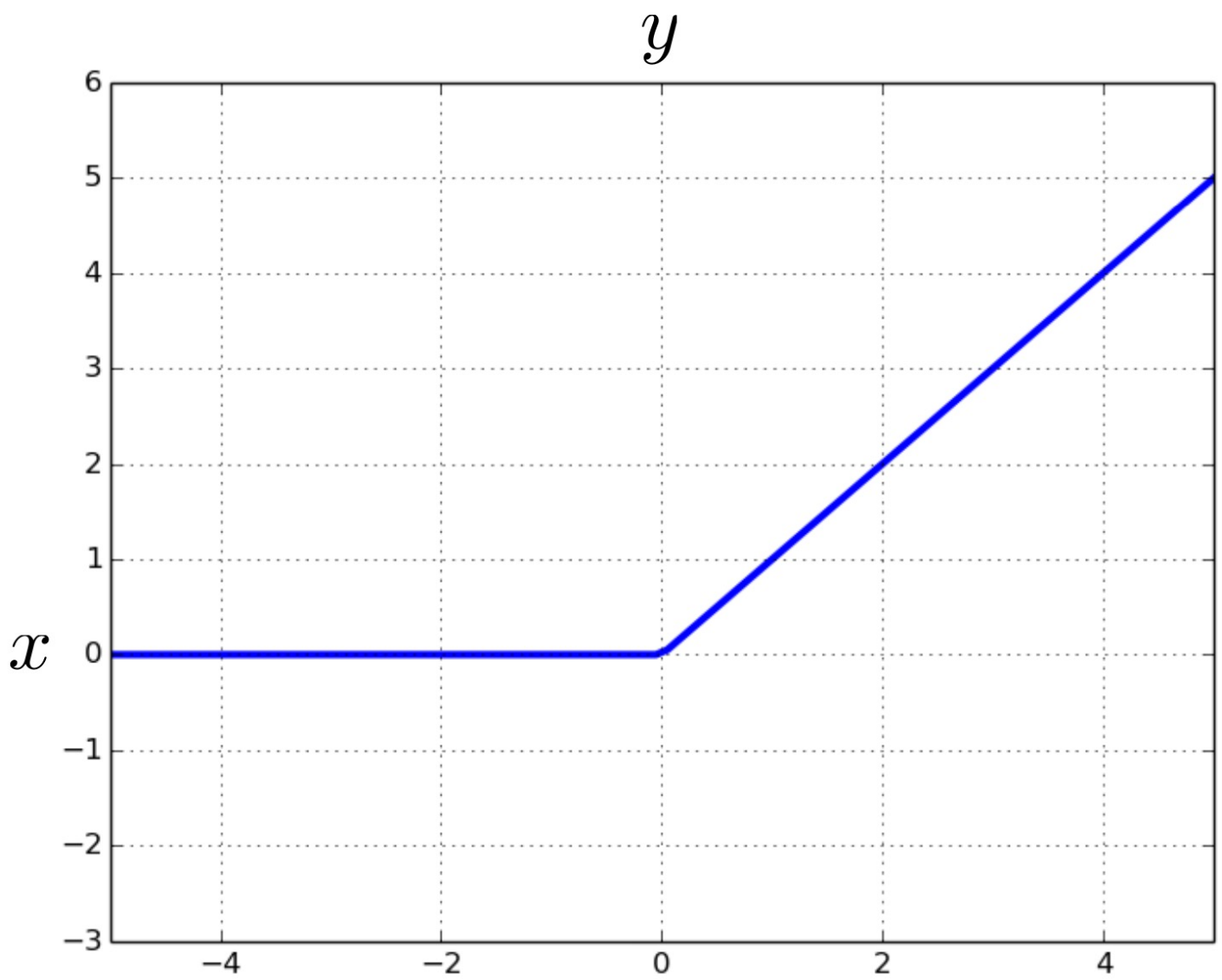
tanh: $y = \tanh(x)$



(logistic) sigmoid: $$y = \frac{1}{1 + \exp\{-x\}}$$



rectified linear unit (ReLU): $y = \max(0, x)$



Adding layers...

$$\mathbf{z}^{(1)} = g\left(\mathbf{U}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$

$$\mathbf{z}^{(2)} = g\left(\mathbf{U}^{(1)}\mathbf{z}^{(1)} + \mathbf{b}^{(1)}\right)$$

$$\ldots$$

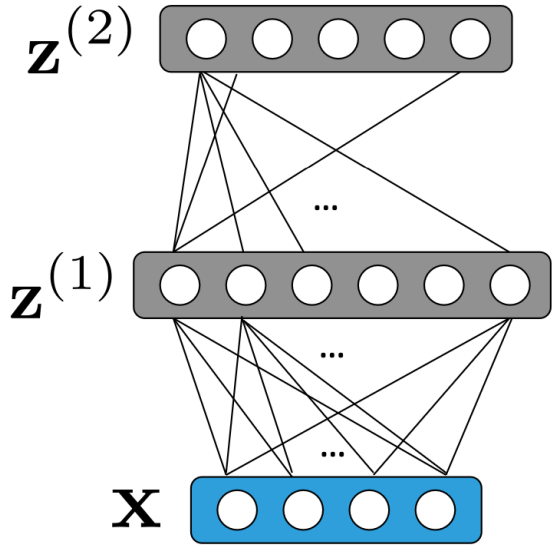
- Use output of one layer as input to next

Adding layers...

$$\mathbf{z}^{(1)} = g\left(\mathbf{U}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$

$$\mathbf{z}^{(2)} = g\left(\mathbf{U}^{(1)}\mathbf{z}^{(1)} + \mathbf{b}^{(1)}\right)$$

$$\ldots$$



- Use output of one layer as input to next
- "feed-forward" and/or "fully-connected" layers

Neural Network for Sentiment Classification

$$\mathbf{z}^{(1)} = g\left(\mathbf{U}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$

$$\mathbf{s} = \mathbf{U}^{(1)}\mathbf{z}^{(1)} + \mathbf{b}^{(1)}$$

vector of label scores

Neural Network for Sentiment Classification

$$\mathbf{z}^{(1)} = g\left(\mathbf{U}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$

$$\mathbf{s} = \mathbf{U}^{(1)}\mathbf{z}^{(1)} + \mathbf{b}^{(1)}$$

$$\mathbf{s} = \begin{bmatrix} \text{score}(\boldsymbol{x}, \text{positive}, \boldsymbol{w}) \\ \text{score}(\boldsymbol{x}, \text{negative}, \boldsymbol{w}) \end{bmatrix}$$

Use softmax function to convert scores into probabilities

$$\mathbf{s} = \begin{bmatrix} \text{score}(\boldsymbol{x}, \text{positive}, \boldsymbol{w}) \\ \text{score}(\boldsymbol{x}, \text{negative}, \boldsymbol{w}) \end{bmatrix}$$

$$\mathbf{p} = \text{softmax}(\mathbf{s}) = \begin{bmatrix} \frac{\exp\{\text{score}(\boldsymbol{x}, \text{positive}, \boldsymbol{w})\}}{Z} \\ \frac{\exp\{\text{score}(\boldsymbol{x}, \text{negative}, \boldsymbol{w})\}}{Z} \end{bmatrix}$$

$$Z = \exp\{\text{score}(\boldsymbol{x}, \text{positive}, \boldsymbol{w})\} + \exp\{\text{score}(\boldsymbol{x}, \text{negative}, \boldsymbol{w})\}$$

Why nonlinearities?

network with 1 hidden layer:

$$\mathbf{z}^{(1)} = g\left(\mathbf{U}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$

$$\mathbf{s} = \mathbf{U}^{(1)}\mathbf{z}^{(1)} + \mathbf{b}^{(1)}$$

- If g is linear, then we can rewrite the above as a single affine transform
- Can you prove this? (use distributivity of matrix multiplication)

Learning with Neural Networks

$$\mathbf{z}^{(1)} = g\left(\mathbf{U}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$

$$\mathbf{s} = \mathbf{U}^{(1)}\mathbf{z}^{(1)} + \mathbf{b}^{(1)}$$

$$\mathbf{s} = \begin{bmatrix} \text{score}(\boldsymbol{x}, \text{positive}, \boldsymbol{w}) \\ \text{score}(\boldsymbol{x}, \text{negative}, \boldsymbol{w}) \end{bmatrix}$$

$$\text{classify}(\boldsymbol{x}, \boldsymbol{w}) = \underset{y}{\text{argmax}} \;\; \text{score}(\boldsymbol{x}, y, \boldsymbol{w})$$

- We can use any of our loss functions from before, as long as we can compute (sub)gradients
- Algorithm for doing this efficiently: **backpropagation**
- Basically just the chain rule of derivatives

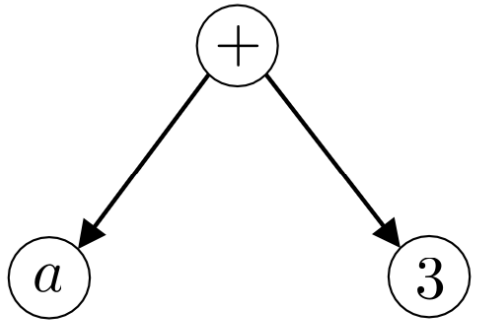
Computation Graphs

- A useful way to represent the computations performed by a neural model (or any model!)
- Why useful? makes it easy to implement automatic differentiation (backpropagation)
- Many neural net toolkits let you define your model in terms of computation graphs (PyTorch, TensorFlow, DyNet, Theano, etc.)

Backpropagation

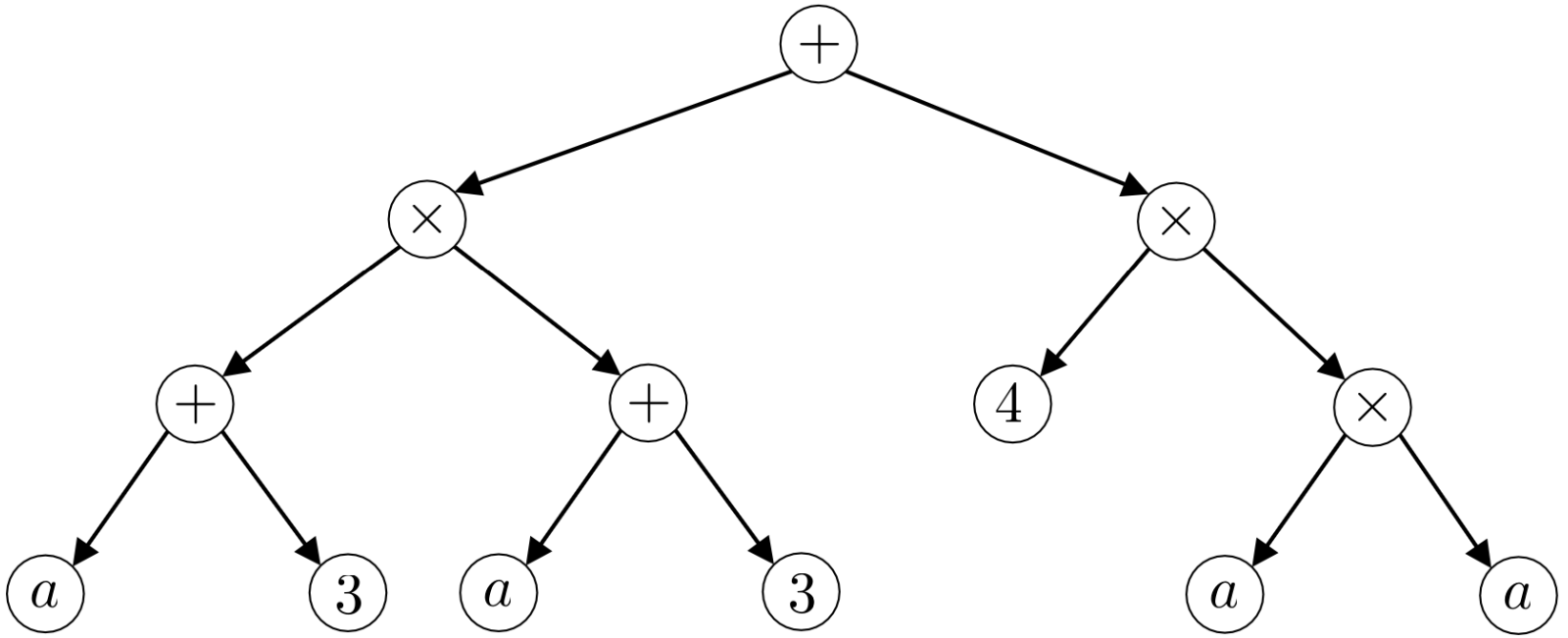
- Backpropagation has become associated with neural networks, but it's much more general
- Can be also used to compute gradients in **linear** models for structured prediction

A simple computation graph:



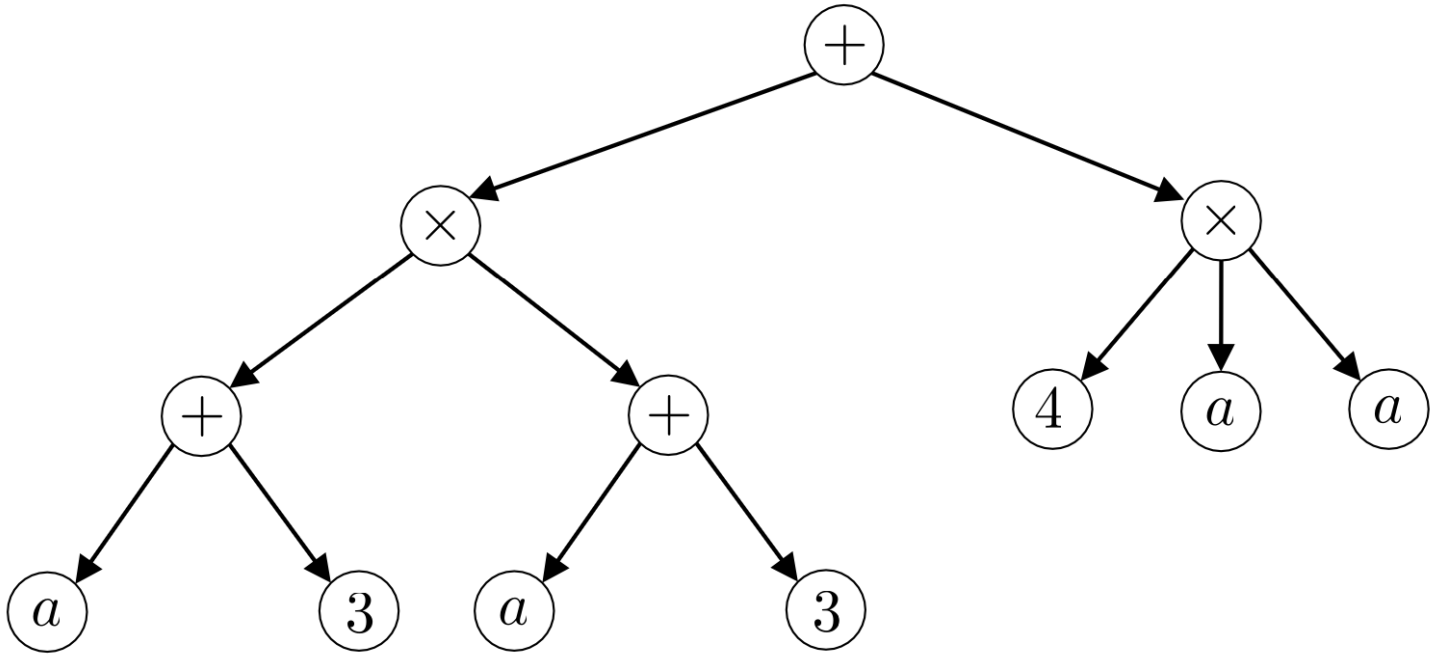
- Represents expression “a + 3”

A slightly bigger computation graph:

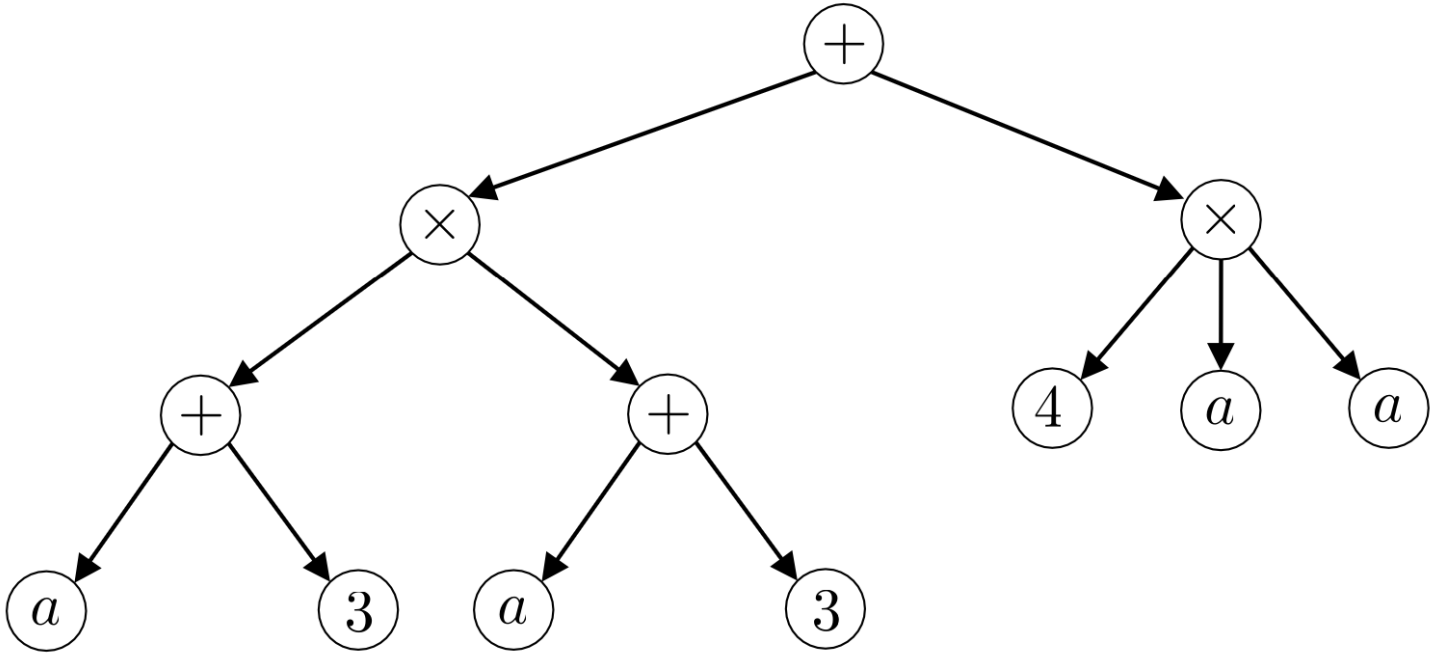


- Represents expression "$(a + 3)^2 + 4a^2$"

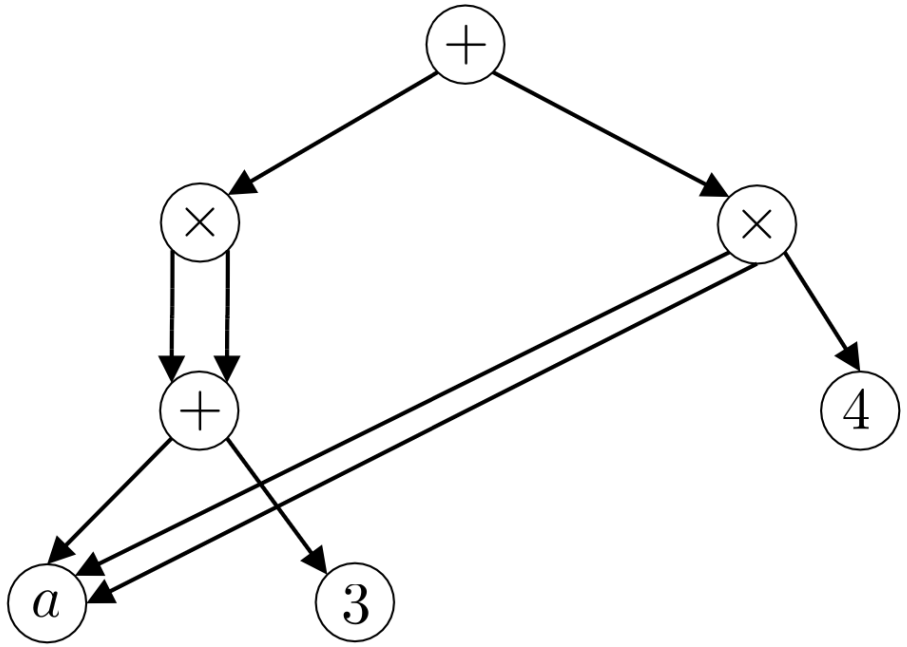
Operators can have more than 2 operands:



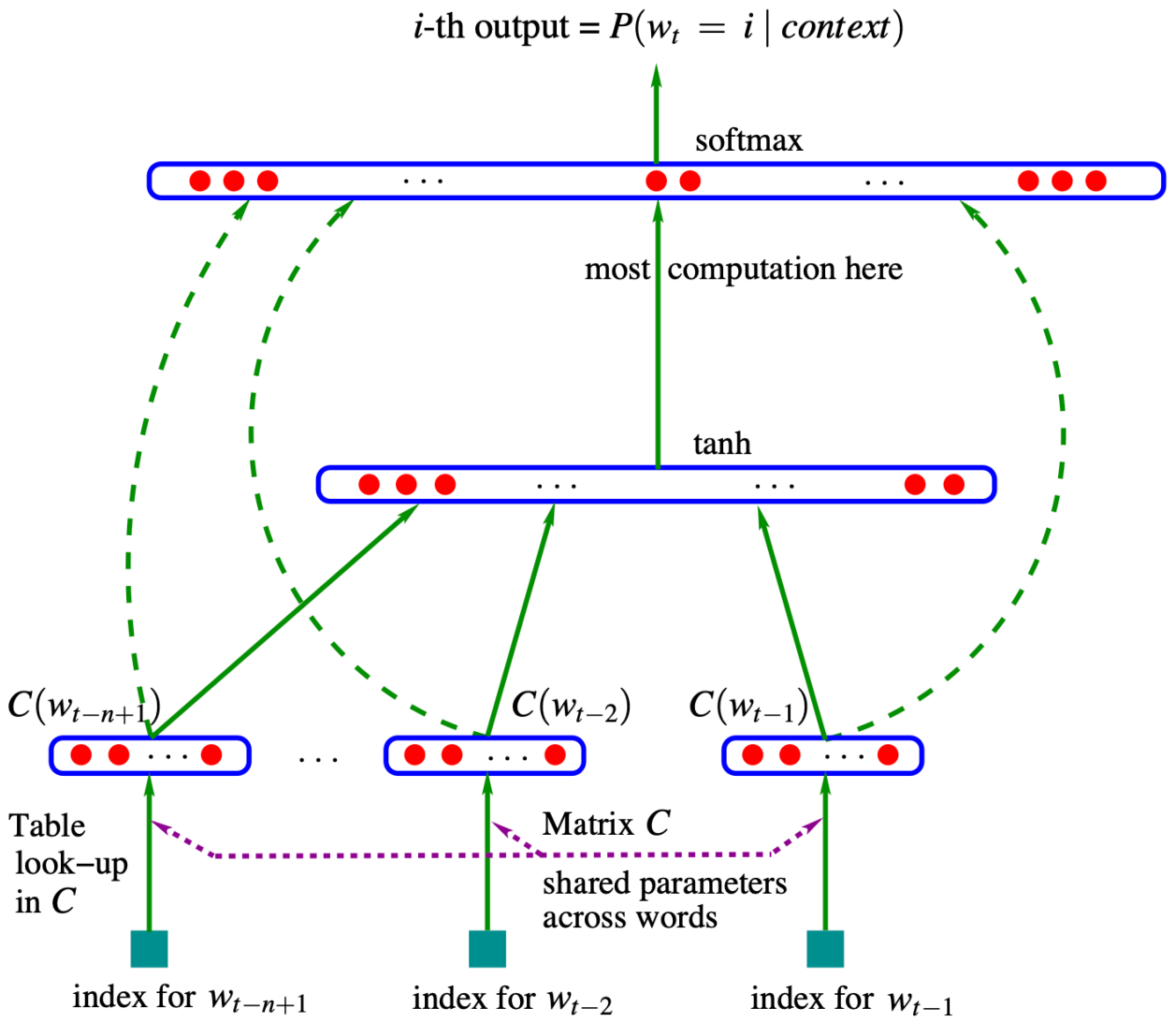
- Still represents expression "$(a + 3)^2 + 4a^2$"



- More concise:

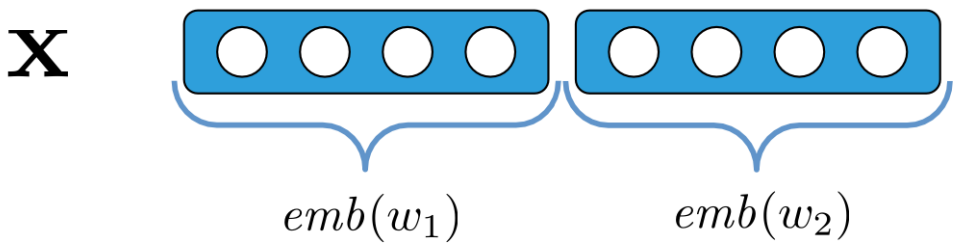


Model (Bengio et al., 2003)



A Simple Neural Trigram Language Model

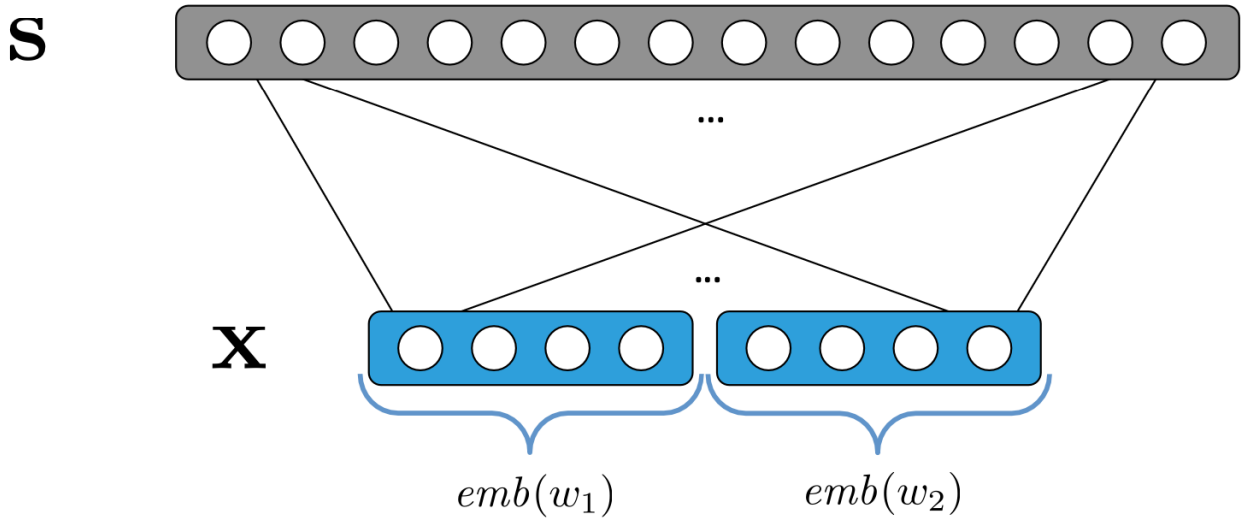
- Given previous words w1 and w2, predict next word
- Input is concatenation of vectors (embeddings) of previous words:



$$\mathbf{x} = cat(emb(w_1), emb(w_2))$$

A Simple Neural Trigram Language Model

- Output vector contains scores of possible next words:



$$emb(w) \in \mathbb{R}^d$$

$$\mathbf{x} \in \mathbb{R}^{2d}$$

$$\mathbf{s} \in \mathbb{R}^{|\mathcal{V}|}$$

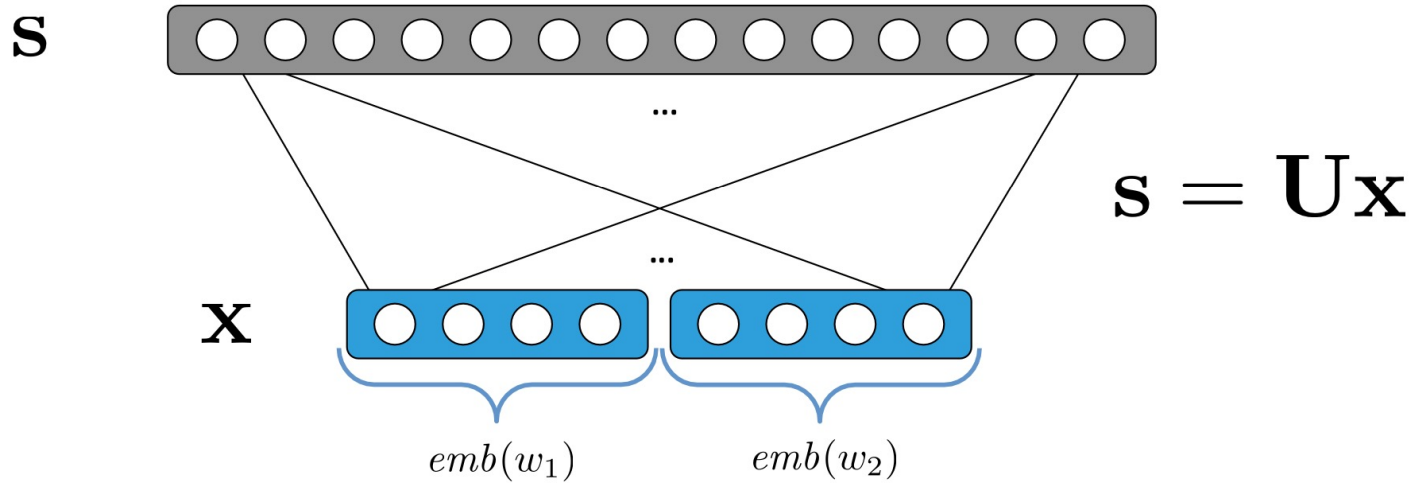
$$\mathbf{U} \in \mathbb{R}^{|\mathcal{V}| \times 2d}$$

$$\mathbf{s} = \mathbf{U}\mathbf{x}$$

$$s_i = \text{score}(\mathbf{x}, w_i, \mathbf{U})$$

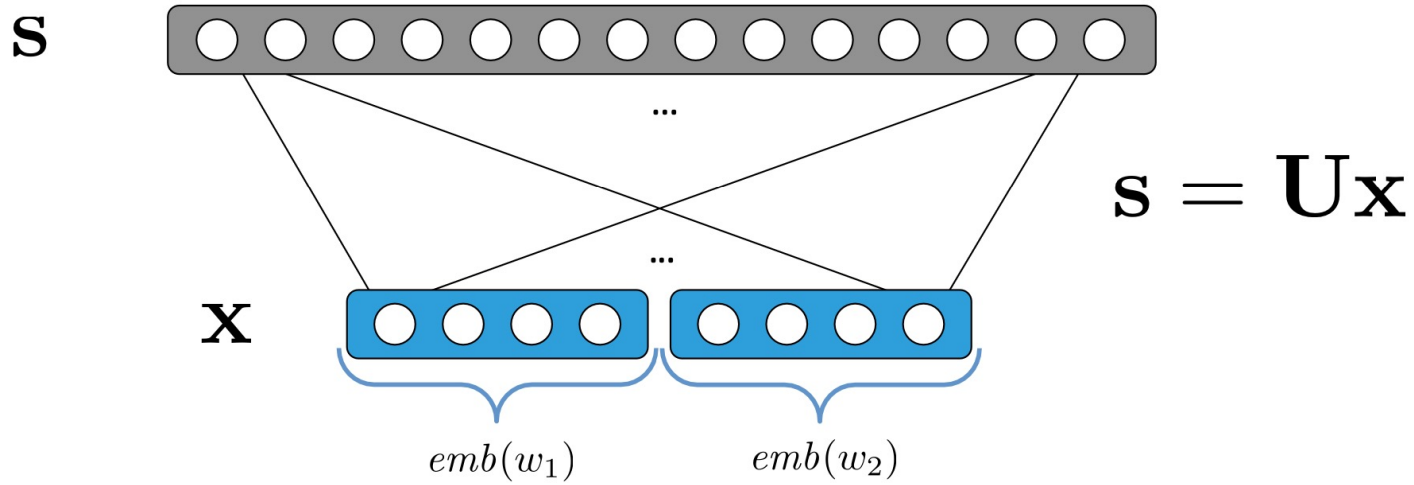
$$\text{score}(\mathbf{x}, w_i, \mathbf{U}) = \mathbf{x}^\top \mathbf{U}_{i,1:d}$$

A Simple Neural Trigram Language Model



- How many parameters are in this model? $|\mathcal{V}| \times 3d$

A Simple Neural Trigram Language Model



- Most common way: log loss

$$\text{loss}_{\log}(\langle w_1, w_2 \rangle, w_3, \boldsymbol{\theta}) = -\log p_{\boldsymbol{\theta}}(w_3 \mid \langle w_1, w_2 \rangle)$$

$$p_{\boldsymbol{\theta}}(w_3 \mid \langle w_1, w_2 \rangle) \propto \exp\{\text{score}(cat(emb(w_1), emb(w_2)), w_3, \mathbf{U})\}$$