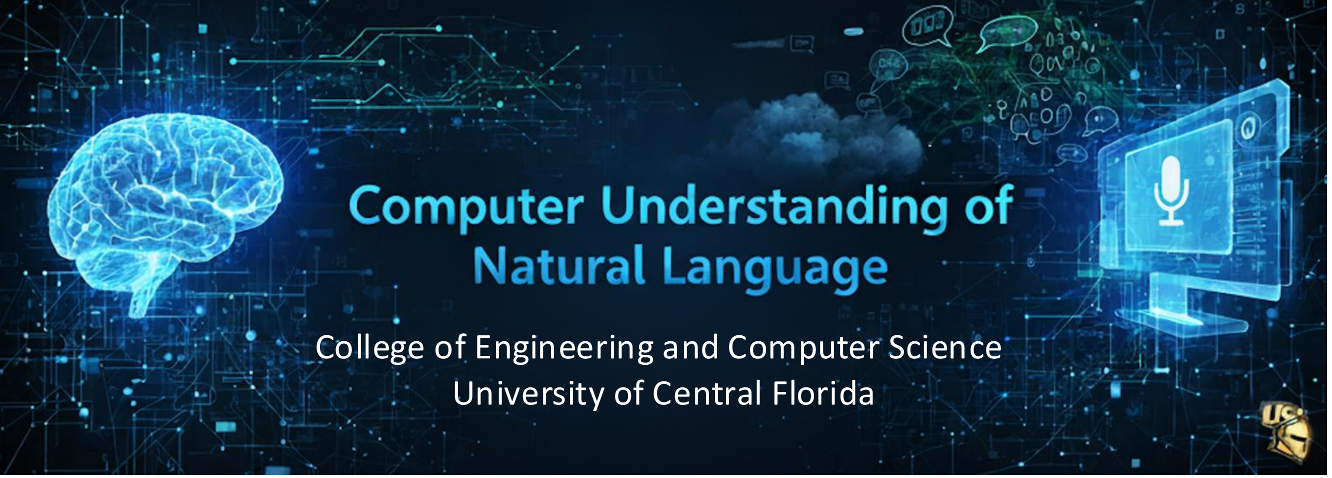


David Mohaisen, Professor
University of Central Florida
Department of Computer Science
Cybersecurity and Privacy Cluster

https://cs.ucf.edu/~mohaisen/

From One-Hot Encoding to Word Embeddings

① Usable meanings in computers
② Representing words
③ Word2Vec overview
④ Word2Vec details
⑤ Gradient descent
⑥ word2Vec example

Objectives

- 1-hot encoding as a computational representation
- Problems with 1-hot encoding, distributional semantics, and word vectors for representation.
- Word2Vec
  - Operation, training and considerations
  - Model optimization
  - Gradient descent and stochastic gradient descent.

Usable Meanings in Computers

- Common solution (WordNet): use a thesaurus containing lists of synonyms sets and hypernyms ("is a" relationships)

e.g. synonym sets containing "good":

```
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()])))
```

```
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
```

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')]
```

Problems with WordNet

- Great as a resource but missing nuance
  - “proficient” is listed as a synonym for “good”.
  - This is only correct in some contexts.
- Missing new meanings of words:
  - wicked, badass, nifty, wizard, genius, ninja, bombast, …; a lot of new words
  - Impossible to keep up-to-date without effort
- Varying degrees of subjectivity
- Requires human labor to create and adapt
- Cannot compute accurate word similarity

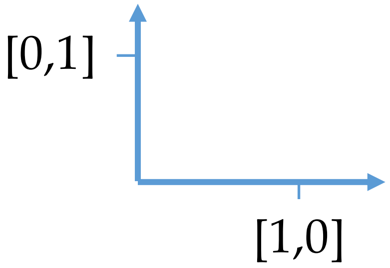
Representing Words as Symbols

- In traditional NLP, we regard words as *discrete* symbols: hotel, conference, motel => a *localist* representation
- Example vocabulary: ["cat", "dog", "fish"]; 1-hot vectors:
  - "cat": [1, 0, 0]
  - "dog": [0, 1, 0]
  - "fish": [0, 0, 1]

- Vector dimension: #vocab (e.g., 500,000)

Problem with Discrete Symbols

- Example: in web search, if user searches for "Seattle motel", we would like to match documents containing "Seattle hotel"; first represent the words:
  - motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0…]
  - hotel = [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0…]



- Observation:

  (1) These vectors are *orthogonal!*

  → There is no natural notion of similarity for one-hot vectors.

- Solution:

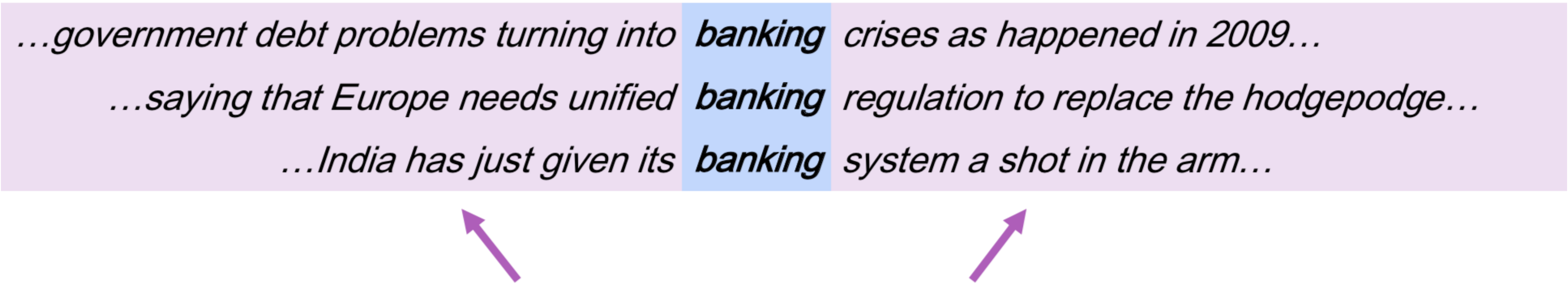
  (1) Could try to rely on WordNet's list of synonyms to get similarity?

  → But then it's known to fail badly; incompleteness is an example

  (2) Learn to encode similarity in the vector itself

Representing Words by Context?

- Distributional semantics:
  - words meaning is given by words frequently appearing close-by.
  - "you shall know a word by the company it keeps" (J. R. Firth)
  - 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
- That can be generalized to an arbitrary dictionary.
- .



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

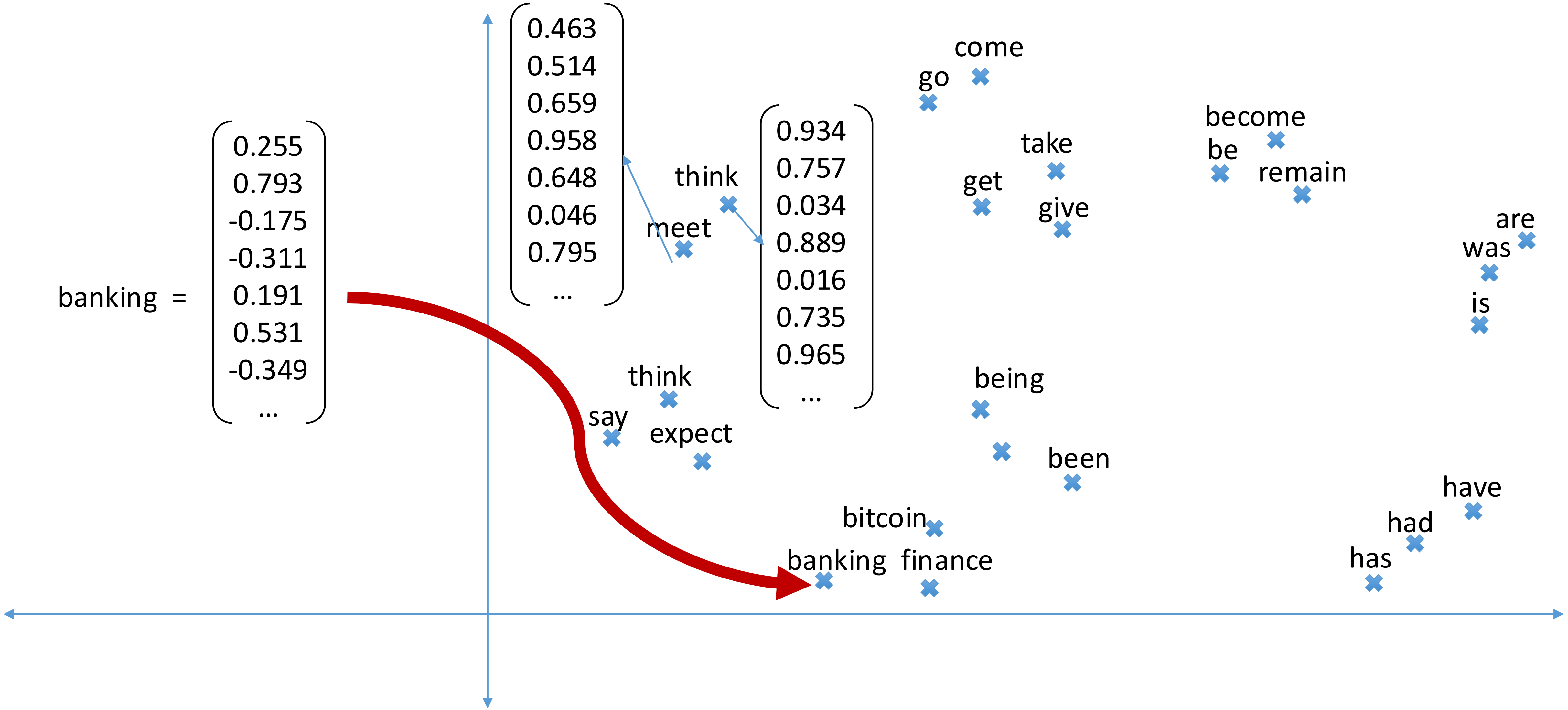
Word Vectors

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

$$\text{banking} = \begin{pmatrix} 0.255 \\ 0.793 \\ -0.175 \\ -0.311 \\ 0.191 \\ 0.531 \\ -0.349 \\ \ldots \end{pmatrix}$$

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

Word Semantic as Neural Vectors

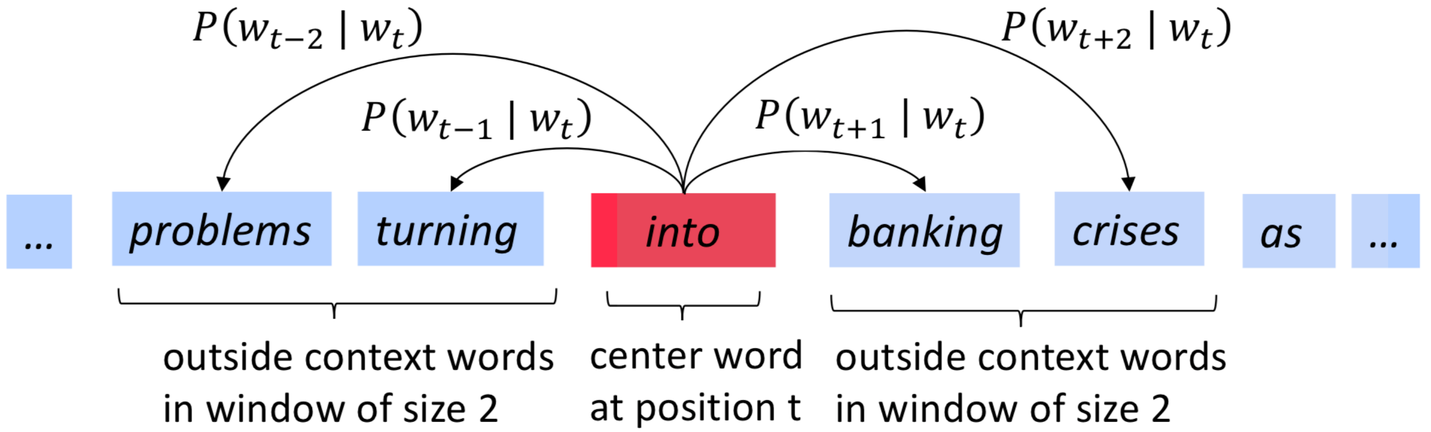


Word2vec: Overview

- Word2vec (Mikolov et al. 2013) is a framework for learning word vectors.
- The main idea of word2vec:
  - We have a large corpus of text
  - Every word is a fixed vocabulary is represented by a vector
  - Go through each position t in the text, which has a center word c and a context ("outside") words o
  - Use the similarity of the word vectors for *c* and o to calculate a probability of o given *c* (or vice versa)
  - Keep adjusting the word vectors to maximize the probability.

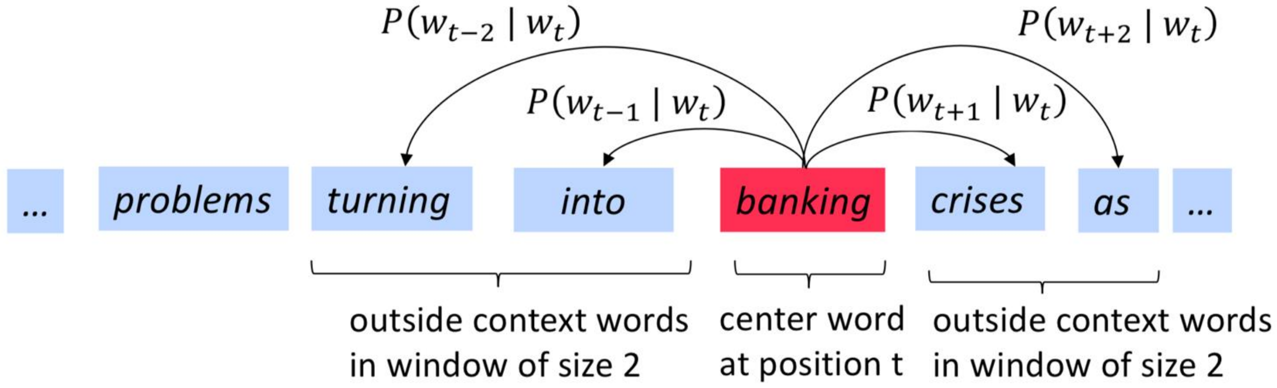
Word2Vec Overview: Example

- Example windows and process for computing $P(w_{t+1} \mid w_t)$.



Word2Vec Overview: Example

- Example windows and process for computing $P(w_{t+1} | w_t)$.



Word2vec: Objective Function

- For each position t=1,… T, predict context words within window of size m, given center word w_j

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

θ is all variables to be optimized

sometimes called *cost* or *loss* function

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

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

- Minimizing objective function ⟷ maximizing predictive accuracy

Word2vec: Objective Function

- We want to minimize the objective function

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

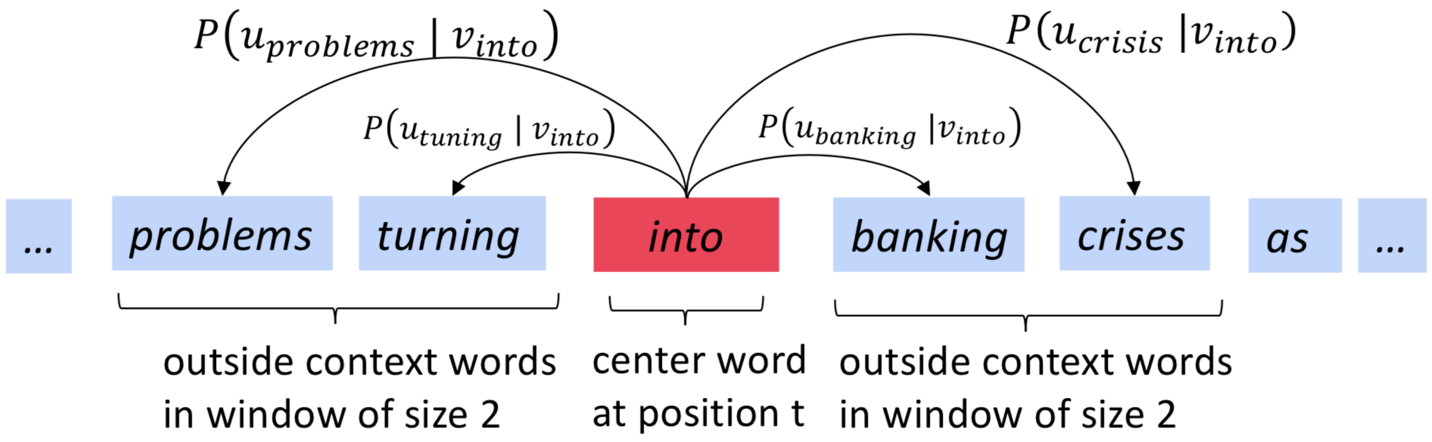
- Question: how to calculate $P(w_{t+j} \mid w_t; \boldsymbol{\theta})$?
- Answer: we will use two vectors per word w
  - v_w when w is a center word
  - u_w when w is a context word
- Then for a center word c and a context o:

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

- .

Word2Vec Overview with Vectors

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



Word2Vec: Prediction Function

Exponentiation makes anything positive

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

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

Normalize over entire vocabulary
to give probability distribution

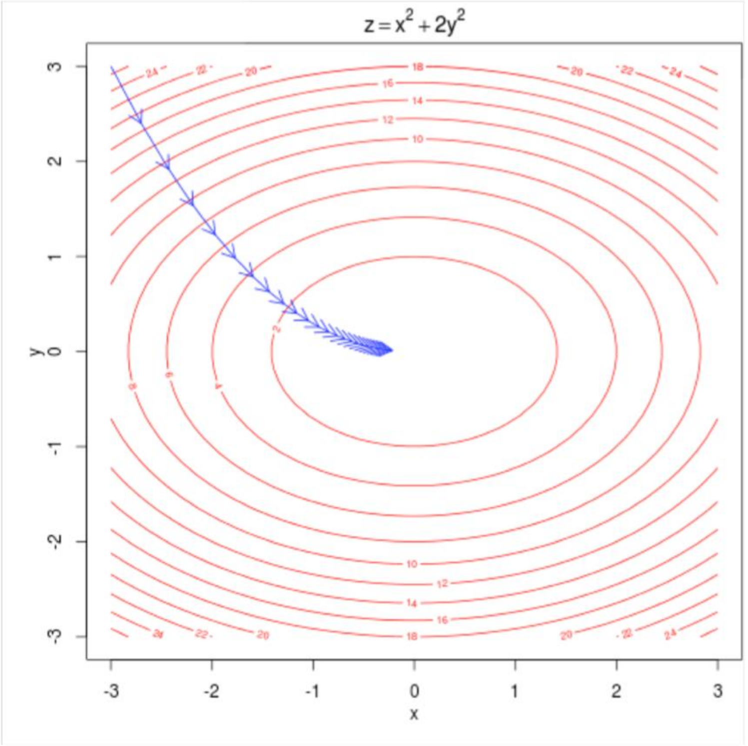
- This is an example of the softmax function $\mathbb{R}^n \to \mathbb{R}^n$
- The softmax function maps an arbitrary input *xi* to a probability distribution p_i

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

  - max because it amplifies the largest xi
  - soft because it still assigns some probability for smaller xi
  - Frequently used in deep learning techniques

Training a Model

- To train a model, we adjust parameters to minimize a loss:
  - Loss here is, for example, error.
  - For a simple convex function over two parameter; contour lines show levels of objective function.



Training a Model by Optimizing

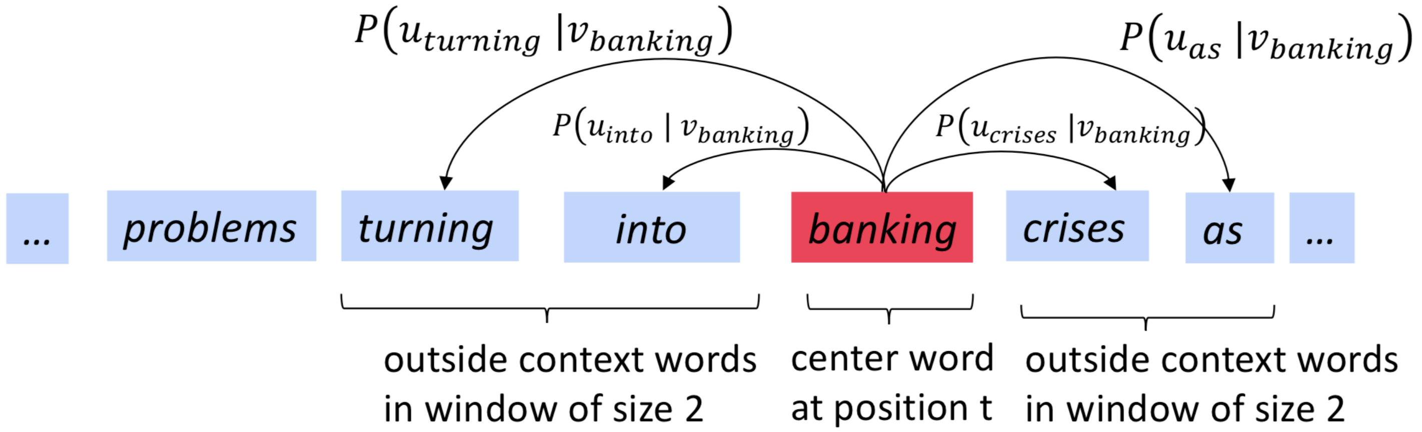
- Compute all vector gradients
  - $\boldsymbol{\theta}$ represents all model parameters; in one long vector
  - In our case with d-dimensional vectors and V-many words:

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

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

Calculating All Gradients

- Generally, in each window we will compute updates for all parameters that are being used in that window. For example

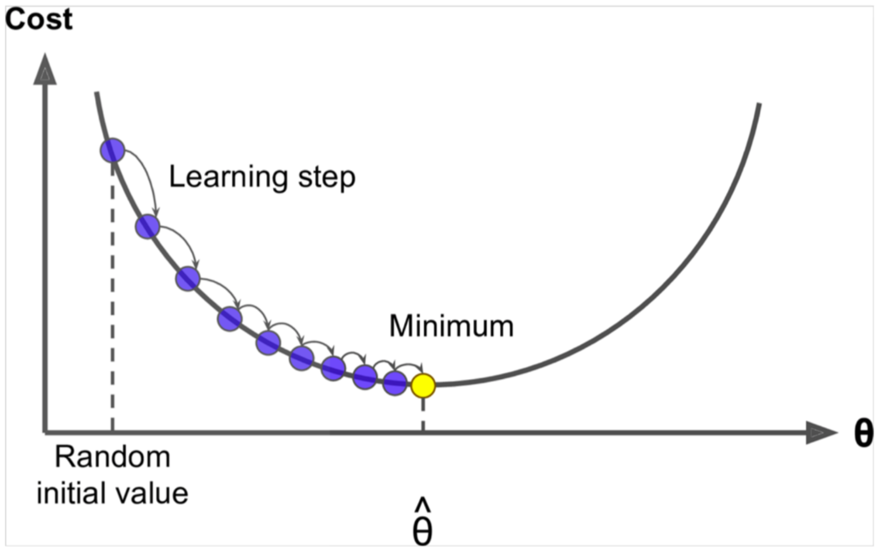


More Details

- Why two vectors ⬄ easier optimization; average both at the end to get better relevance.
- The two model variants:
  - skip-gram: predict context (“outside”) words (position independent) given center word.
  - Continuous bag of words (CBOW): predict center word from (bag of) context words.
- Additional efficiency in training
  - Negative sampling

Optimization: Gradient Descent

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



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):

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

Stochastic Gradient Descent

- Problem: $J(\boldsymbol{\theta})$ is a function of all windows in the corpus (potentially billions)
  - So the update of $J(\boldsymbol{\theta})$ over $\boldsymbol{\theta}$ is expensive
- You would wait a very long time before making a single update.
  - Very bad idea for pretty much all problems
- Solution: SGD: repeatedly sample windows, and update after each one.

Word2Vec: Simply

- **Objective:** Convert a text corpus into a format for training Word2Vec.
  - **Text Corpus:** Start with a large collection of text (e.g., "The cat sat on the mat").
  - **Vocabulary Creation:** Identify the unique words in the corpus to create a vocabulary. Example:
    - Vocabulary =[“the”, ”cat”, “sat”, “on”, “mat”]
- **Context Window:** Define a context window size (*n* words before and after the target word). For example, with a window size of 2:
  - Target word: "sat"
  - Context: ["the", "cat", "on", "the"]

Word2Vec: Simply, cont.

- **Choose a Word2Vec Model**
  - Word2Vec has two main architectures:

  (1) Skip-Gram: Predicts context words given a target word.

  → Example: For the word "sat," predict "cat" and "on."

(2) CBOW (Continuous Bag of Words): Predicts the target word given context words.

→ Example: For context ["the", "cat", "on", "the"], predict "sat."

Word2Vec: Simply, cont.

- **Create Training Data**
  - For each target word, generate (input, output) pairs:

  (1) Skip-Gram Example:
    - Input: "sat"
    - Outputs: "cat", "on"

  (2) CBOW Example:
    - Inputs: "the", "cat", "on", "the"
    - Output: "sat"
- These pairs will be used to train the Word2Vec model.

- **Initialize Word Embedding Matrix**
  - Create two matrices: **Input Embedding Matrix** and **Output Embedding Matrix**.
  - The size of each matrix is (Vocabulary Size × Embedding Dimension).

    → If vocabulary size = 5 and embedding dimension = 3, the matrices will be 5 × 3.

Word2Vec: Simply, cont.

- **Train the Neural Network -- Architecture:** A shallow neural network is used with:
  - **Input Layer:** One-hot encoded representation of the target word.
  - **Hidden Layer:** Produces the word embedding (dense vector).
  - **Output Layer:** Predicts probabilities of context words using a softmax function.

- **Training Steps:**
  - **Forward Pass:**
    - Multiply one-hot vector with the input embedding matrix to get the embedding for the target word.
    - Pass the embedding through the hidden layer.
    - Multiply the hidden layer output with the output embedding matrix to predict context words.
    - Apply softmax to get probabilities of all words in the vocabulary.
  - **Loss Calculation:** Compute the loss using the cross-entropy between predicted and actual context words.
  - **Backpropagation:** Adjust weights of input/output embedding matrices to minimize loss.

Word2Vec: Simply, cont.

- For a vocabulary of size 3 (["cat", "dog", "mat"]) and an embedding size of 2:
  - Input: One-hot encoding of "cat" → [1, 0, 0]
  - Forward pass maps this through the embedding matrix to produce:
    - "cat" → [0.25, 0.75]
    - "dog" → [0.60, 0.80]
    - "mat" → [0.10, 0.90]

Takeaway

(1) Word2Vec is a model that represents words as dense vectors, capturing semantic and syntactic relationships based on context.

(2) Unlike sparse 1-hot encoding, it uses a neural network to learn embeddings through Skip-Gram or CBOW, placing similar words close in a continuous space.

(3) This very idea enhances NLP tasks like sentiment analysis and text classification by creating clear similarity measures.