6.S079 EMBEDDINGS, RAG, AND VECTOR DATABASES

MARCH 14, 2024

MIKE CAFARELLA

AGENDA

1. Embeddings
2. Applications
3. Retrieval Augmented Generation
4. Vector Databases

Word embeddings

- **Idea:** learn a high-dimensional representation of each word

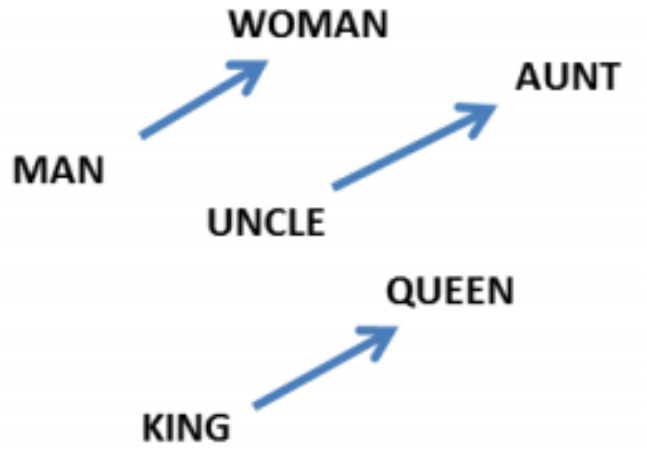
  Cat: {0.002, 0.244, 0.546, …, 0.345}

- Need to have a function W(word) that returns a vector encoding that word.

- Applications: ???

Word embeddings: properties

Relationships between words correspond to difference between vectors.



$$W(\text{“woman”}) - W(\text{“man”}) \simeq W(\text{“aunt”}) - W(\text{“uncle”})$$

$$W(\text{“woman”}) - W(\text{“man”}) \simeq W(\text{“queen”}) - W(\text{“king”})$$

http://colah.github.io/posts/2014-07-NLP-RNNs-Representations/

Word embeddings: questions

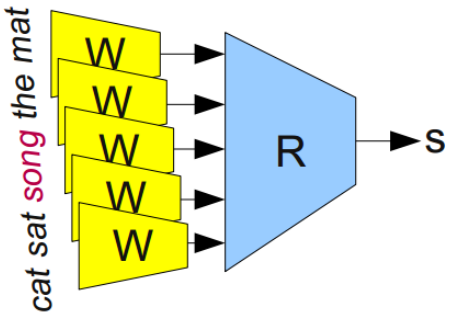
- How big should the embedding space be?
  - Trade-offs like any other machine learning problem – greater capacity versus efficiency and overfitting.

- How do we find W?
  - Often as part of a prediction or classification task involving neighboring words.

Learning word embeddings

https://arxiv.org/ftp/arxiv/papers/1102/1102.1808.pdf

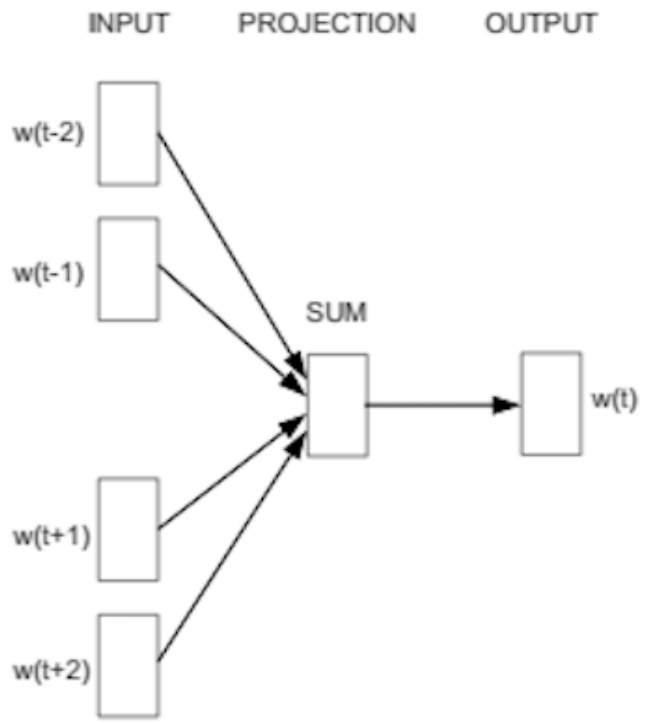
- First attempt:
  - Input data is sets of 5 words from a meaningful sentence. E.g., "one of the best places". Modify half of them by replacing middle word with a random word. "one of function best places"
  - W is a map (depending on parameters, Q) from words to 50 dim'l vectors.
  - Feed 5 embeddings into a module R to determine 'valid' or 'invalid'
  - Optimize over Q to predict better



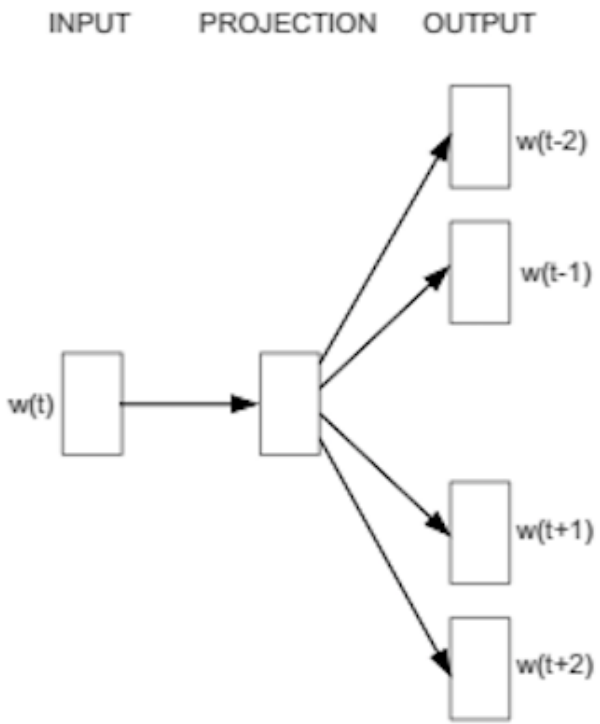
http://colah.github.io/posts/2014-07-NLP-RNNs-Representations/

word2vec

- A set of (relatively) simple methods to produce word embeddings
- Predict words using context
- Two versions: CBOW and Skip-gram



CBOW



Skip-gram

Bag of Words (BOW)

- Bag of words (BOW)
  - A vector representation of word frequency
  - Vector has to be as long as vocabulary size
  - Very old method
  - Gets rid of word order!
- Exploits the "distributional hypothesis"
  - The degree of semantic similarity between two words reflects the similarity of the linguistic contexts in which those words can appear
- Words are points in an n-dimensional boolean space

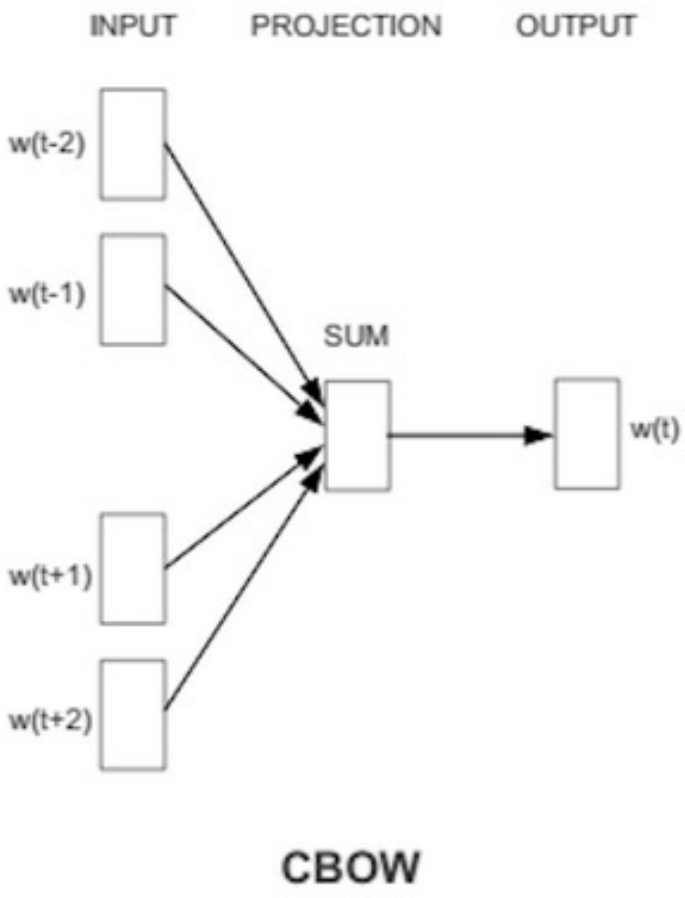
"Cat jumped on dog" [1, 0, 0, 1, …. 1, 1, 0]

"Dog jumped on cat" [1, 0, 0, 1, …. 1, 1, 0]

What are the weaknesses with this approach?

Continuous Bag of Words (CBOW)

- Takes vector embeddings of n words before target and n words after and adds them (as vectors).
- It is a "fill-in-the-blank" task
- The embedding describes how the "missing word" impacts the probability of seeing the words in its context window
- It works well even though we remove word order. The vector sum is meaningful enough to deduce missing word.
- Key idea: **Use combined representations of contextual words to predict the missing word**



Continuous Bag of Words - Window Size 2

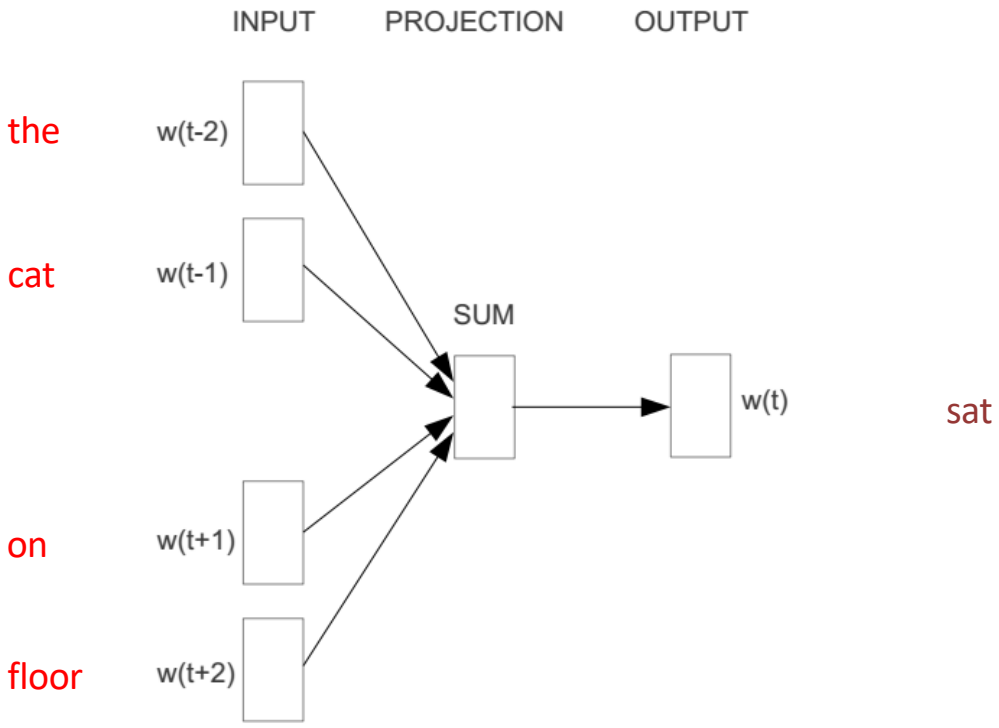
Jay was hit by a _____ bus in...

by	a	red	bus	in

input 1	input 2	input 3	input 4	output
by	a	bus	in	red

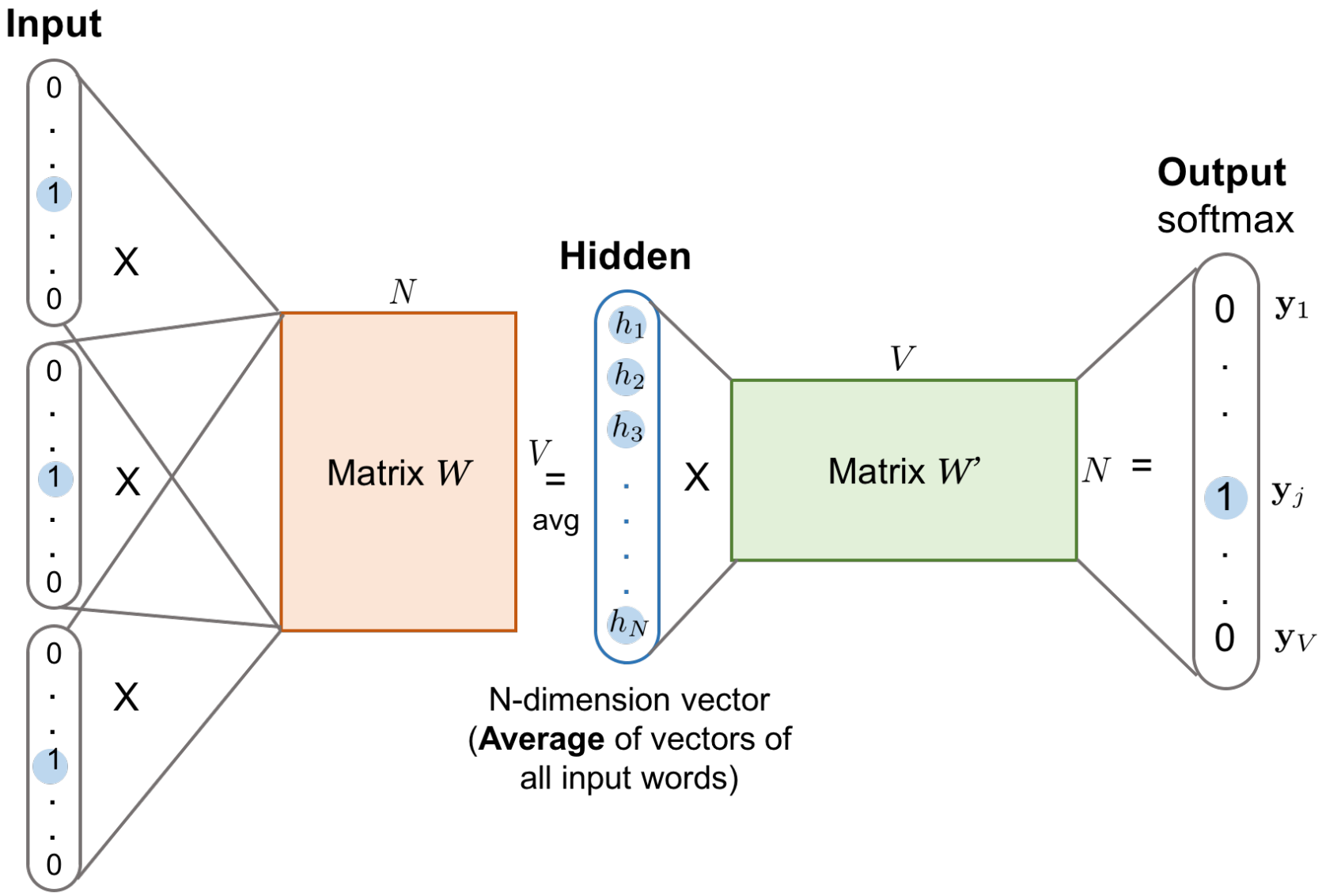
Continuous Bag of Word

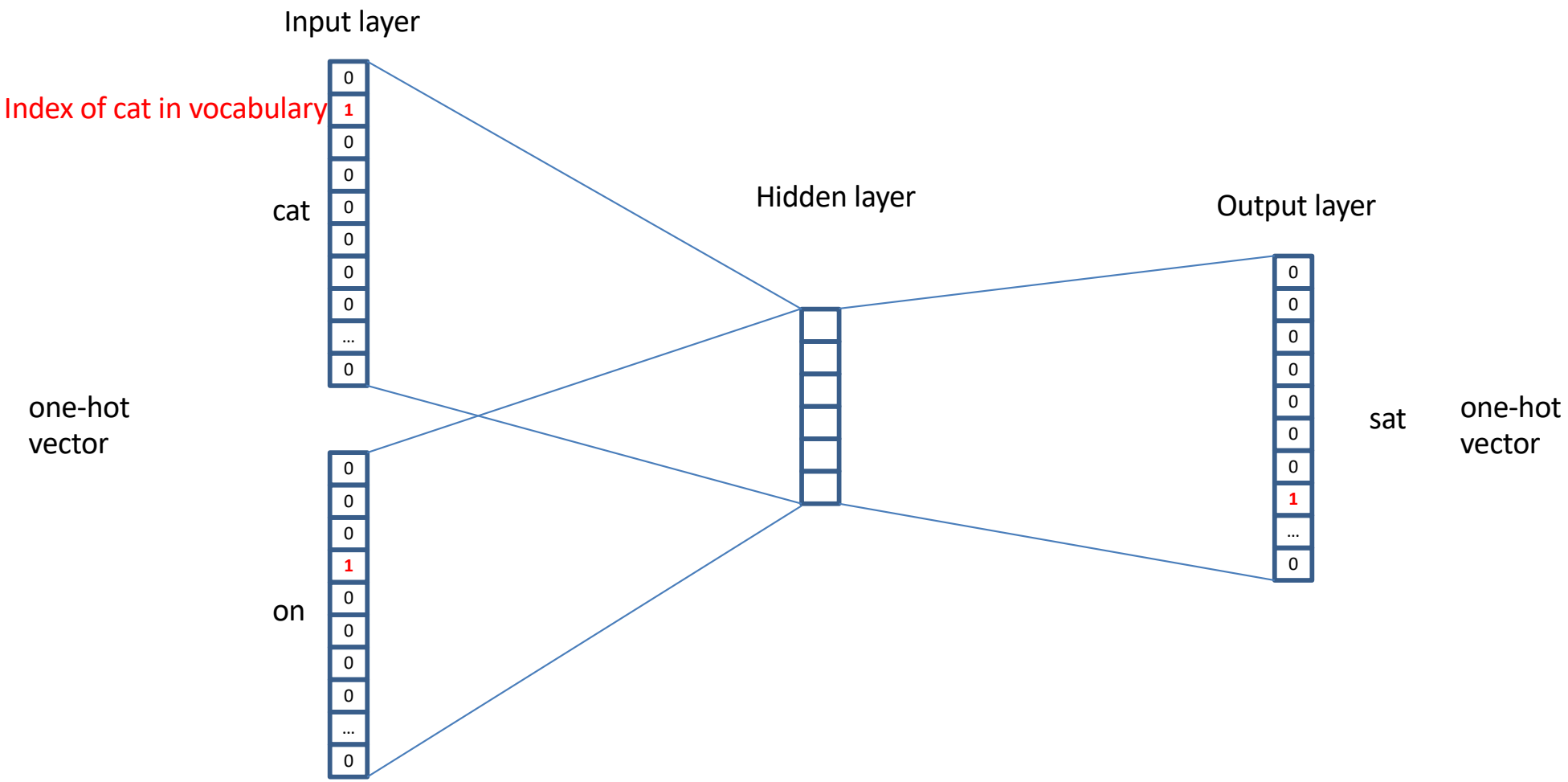
- E.g. “The cat sat on floor”
  - Window size = 2



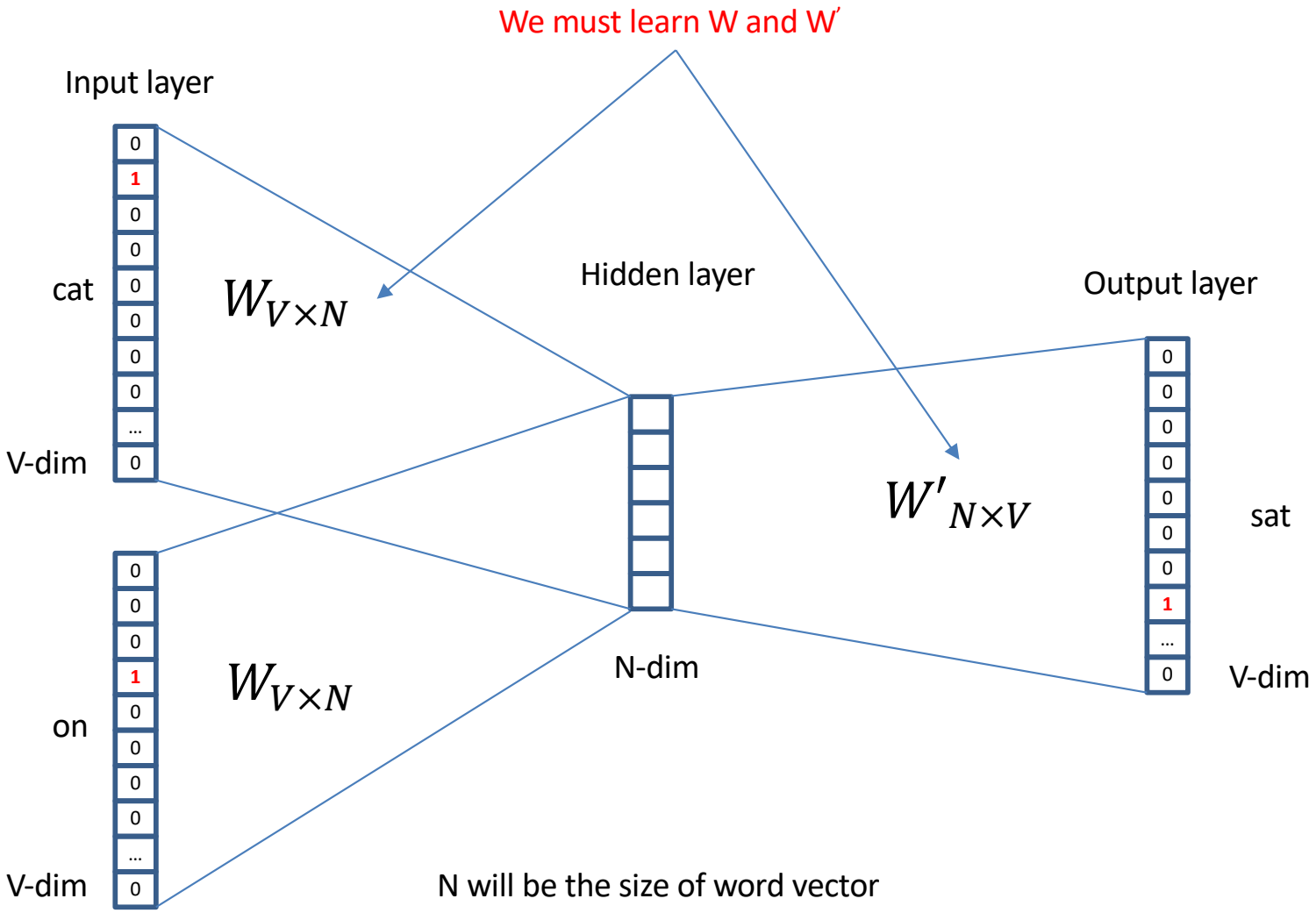
www.cs.ucr.edu/~vagelis/classes/CS242/slides/word2vec.pptx

COBW

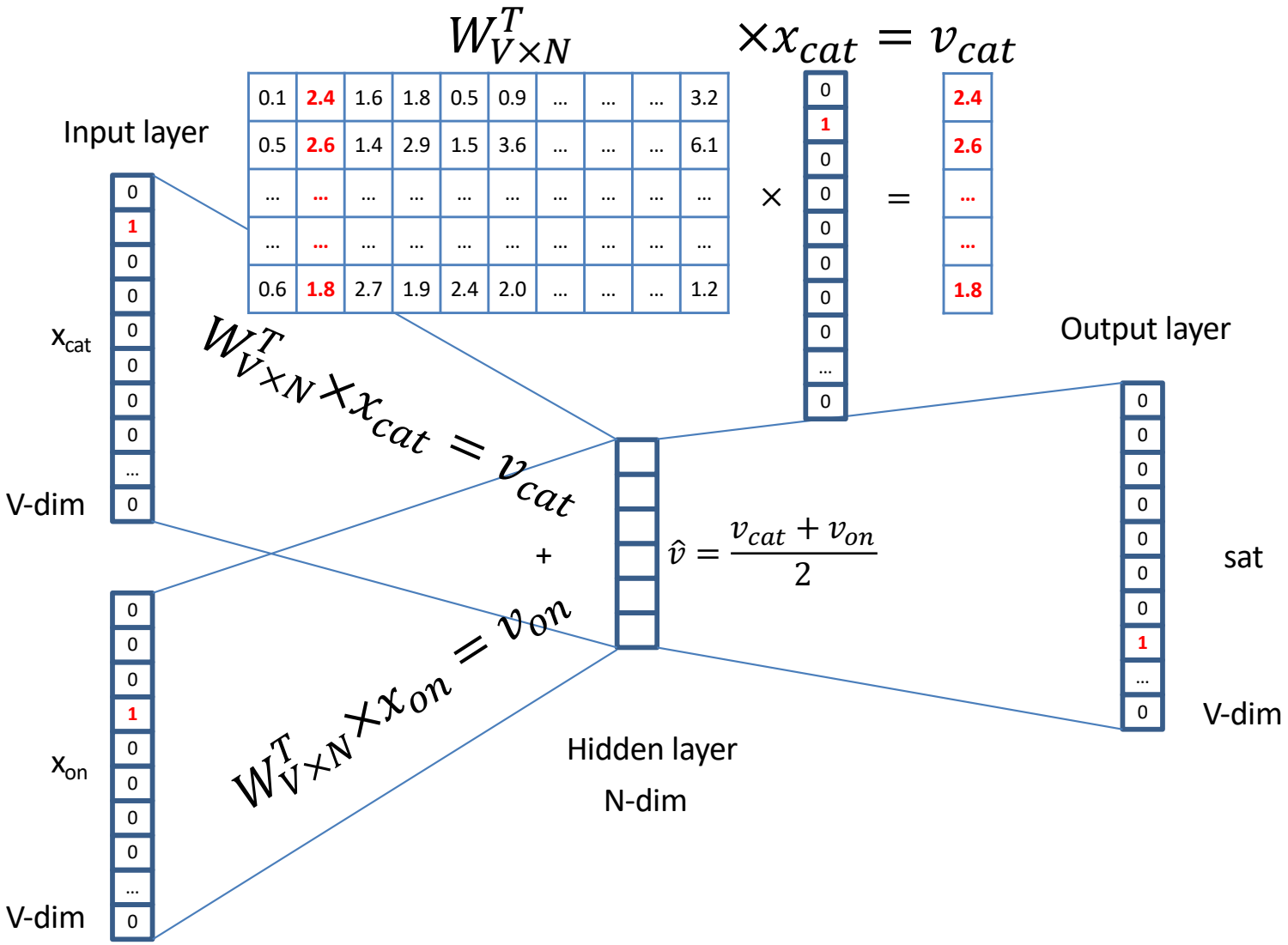

Input
Output
softmax
Hidden
Matrix W
Matrix W'
Matrix W
N
V
V
=
avg
X
X
X
N =
N-dimension vector
(Average of vectors of
all input words)



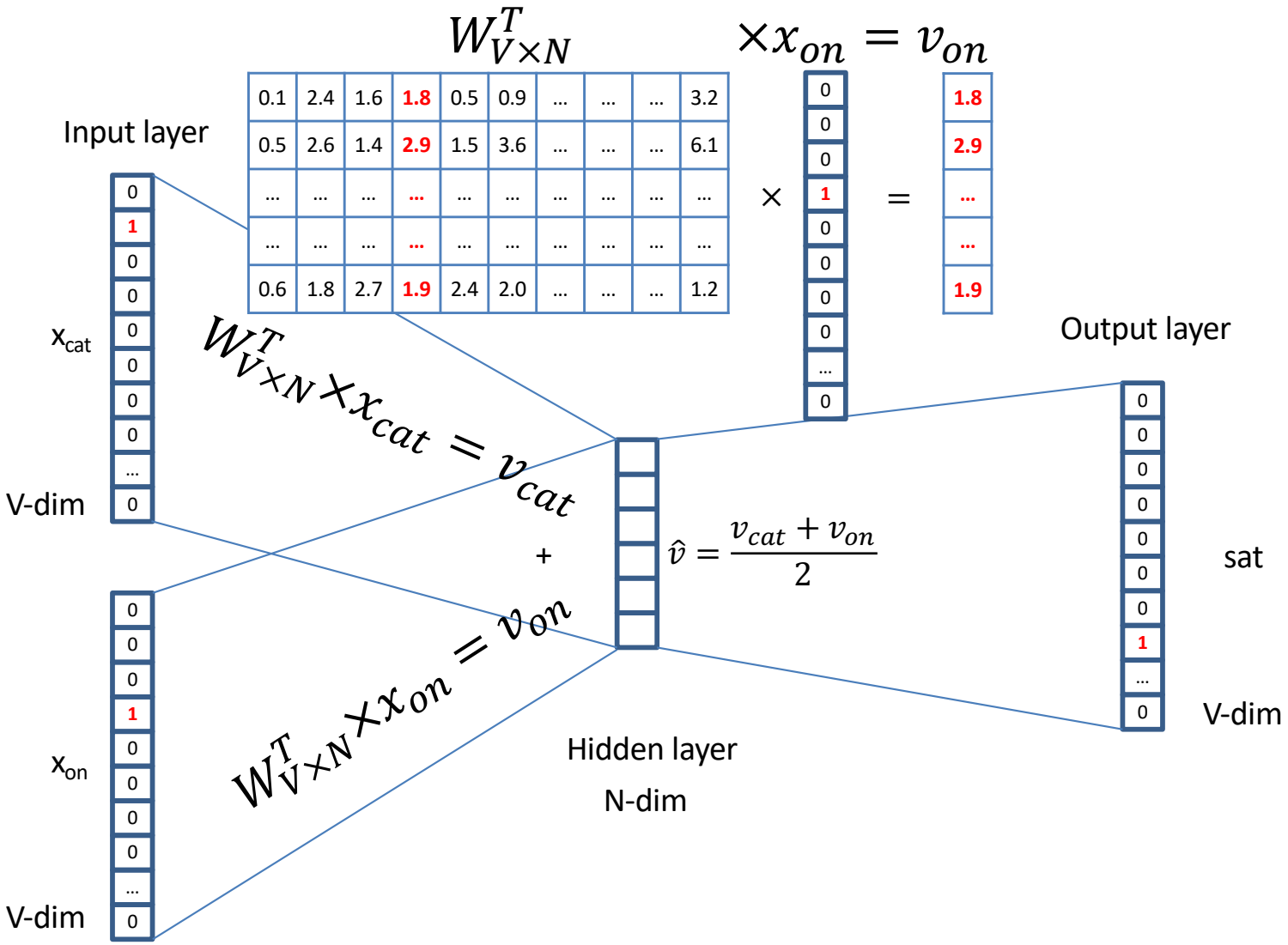
www.cs.ucr.edu/~vagelis/classes/CS242/slides/word2vec.pptx



www.cs.ucr.edu/~vagelis/classes/CS242/slides/word2vec.pptx



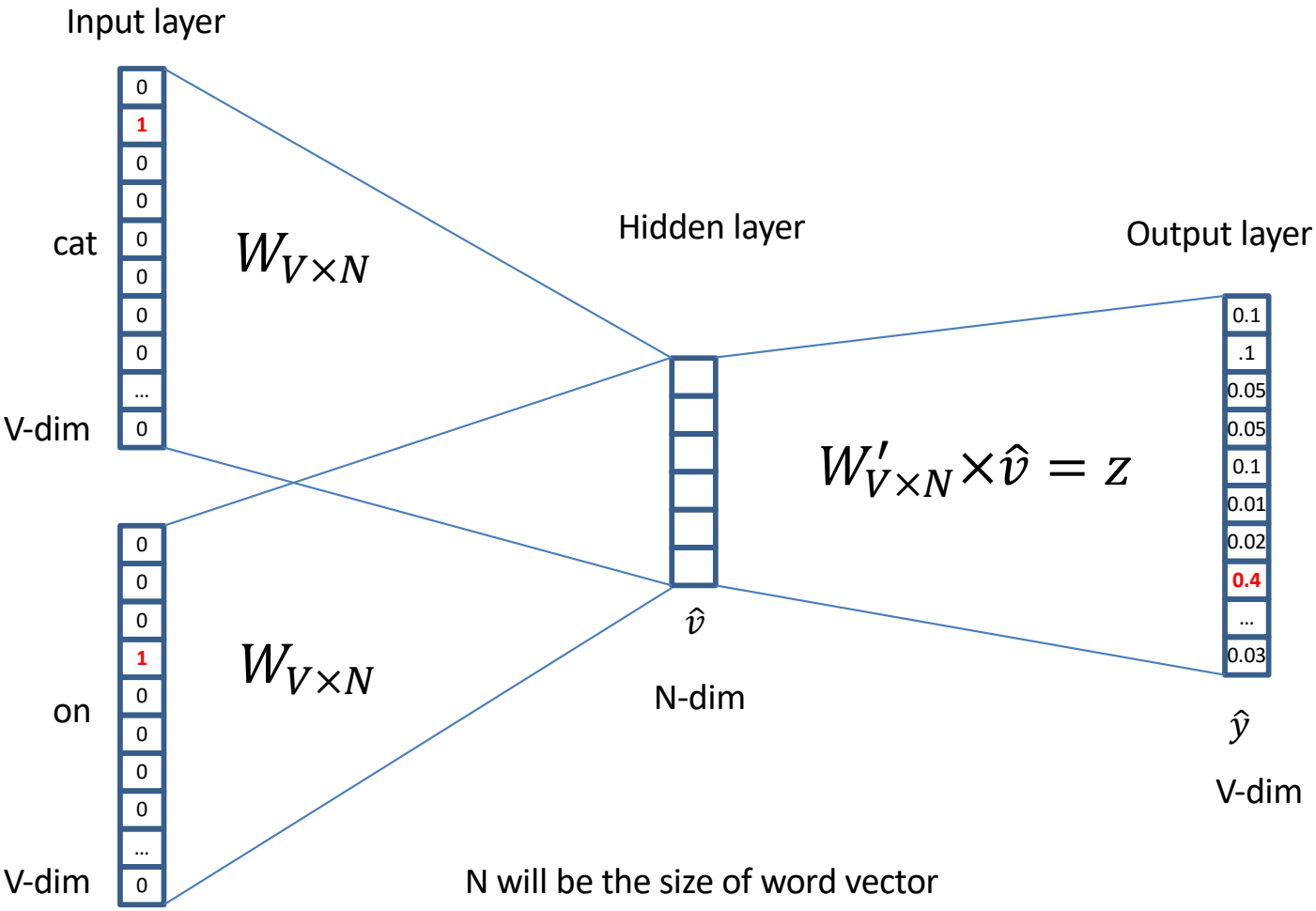
www.cs.ucr.edu/~vagelis/classes/CS242/slides/word2vec.pptx



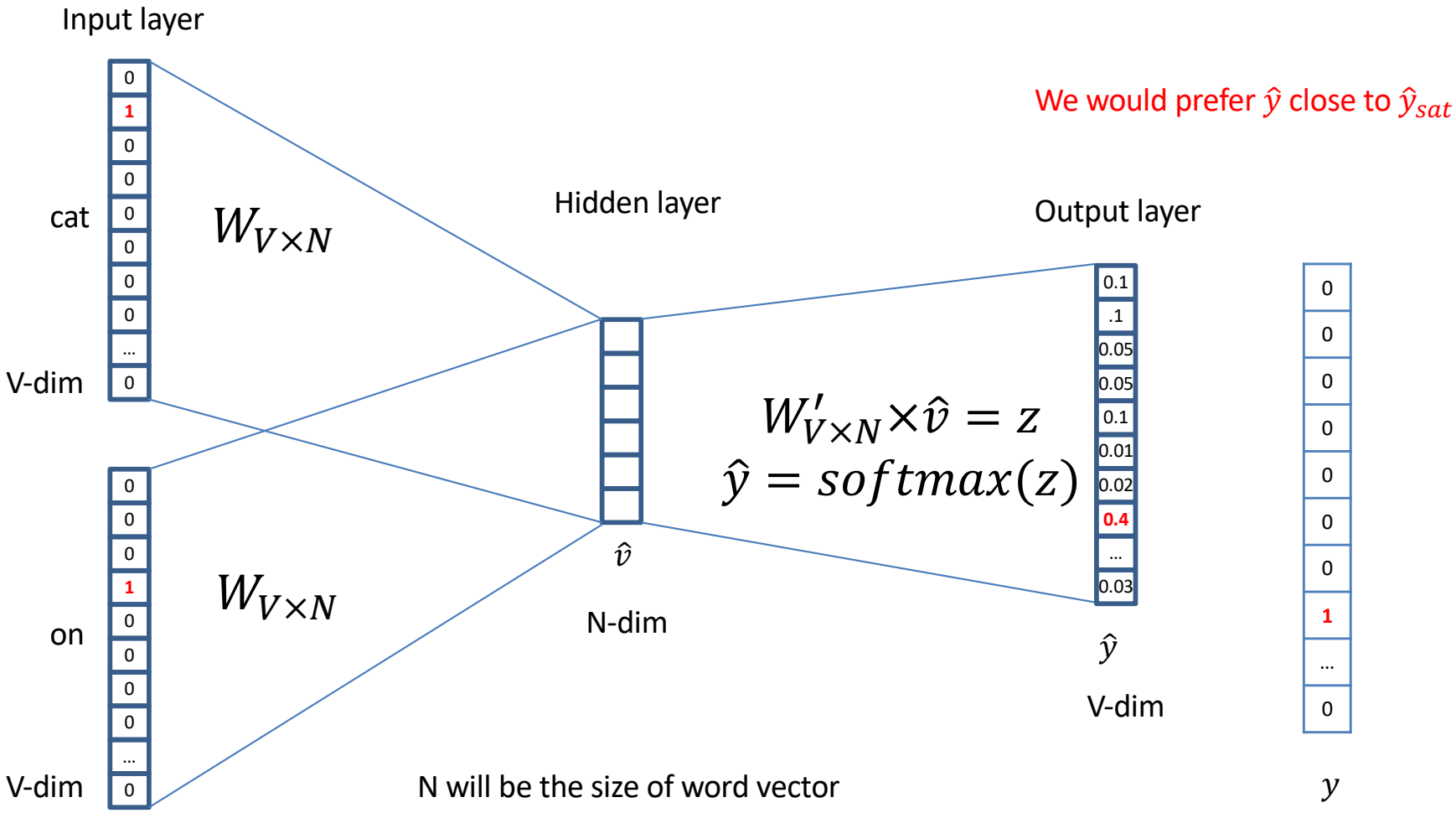
www.cs.ucr.edu/~vagelis/classes/CS242/slides/word2vec.pptx

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

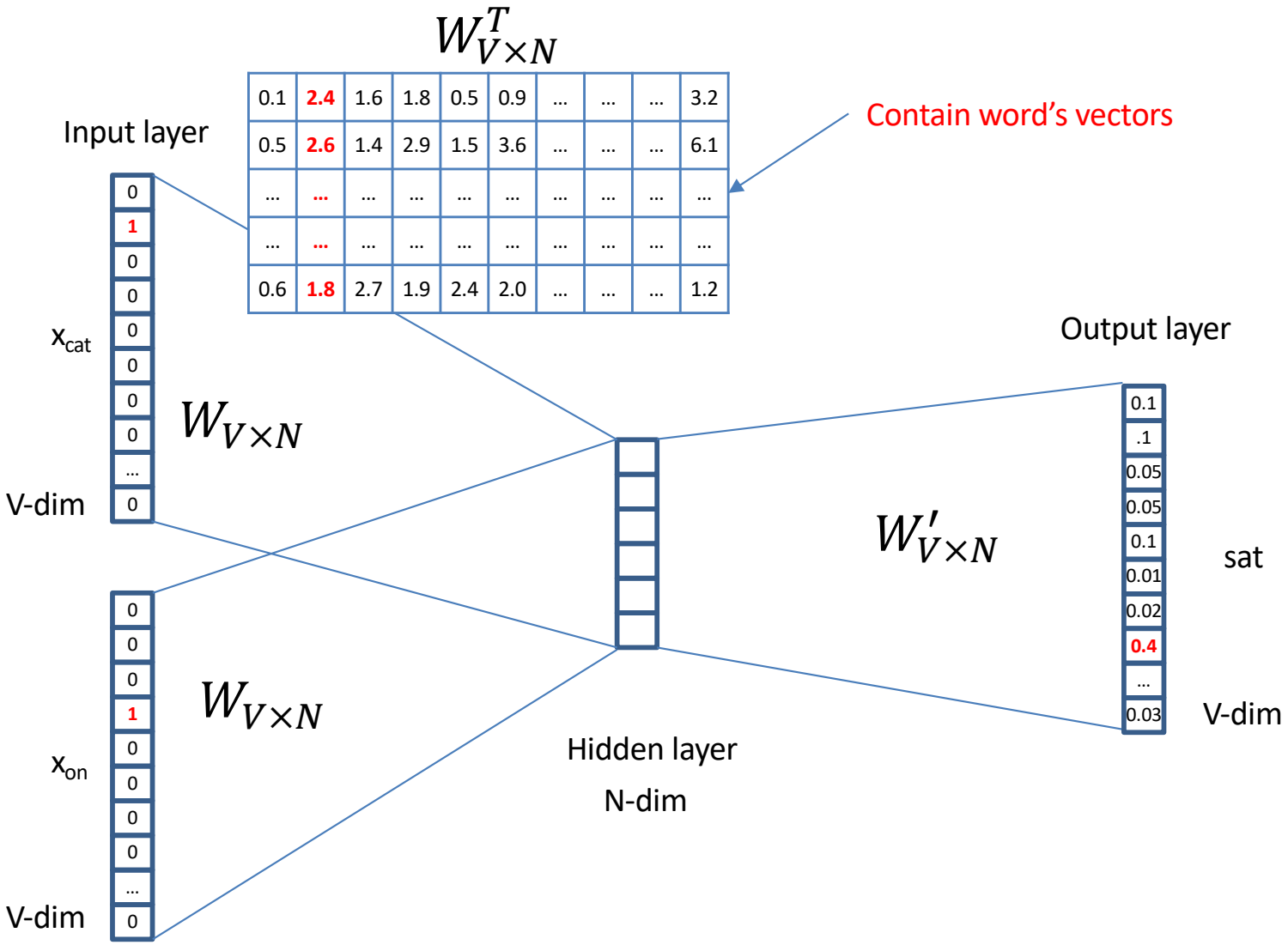
Softmax turns the vector into proabilities



www.cs.ucr.edu/~vagelis/classes/CS242/slides/word2vec.pptx



www.cs.ucr.edu/~vagelis/classes/CS242/slides/word2vec.pptx



We can consider either W or W' as the word's representation. Or even take the average.

www.cs.ucr.edu/~vagelis/classes/CS242/slides/word2vec.pptx

Word Analogies

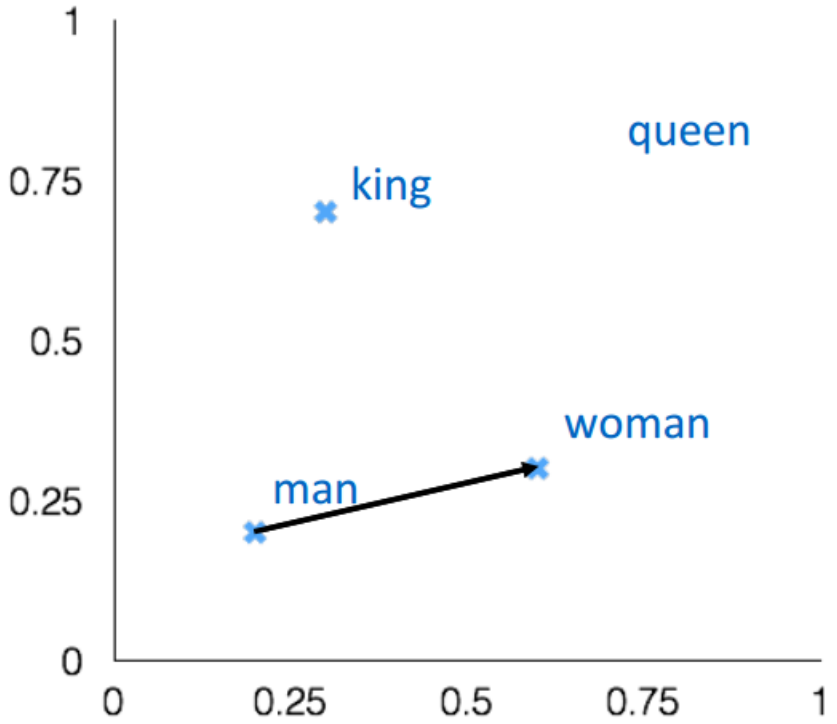
Test for linear relationships, examined by Mikolov et al. (2014)

a:b :: c:?

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

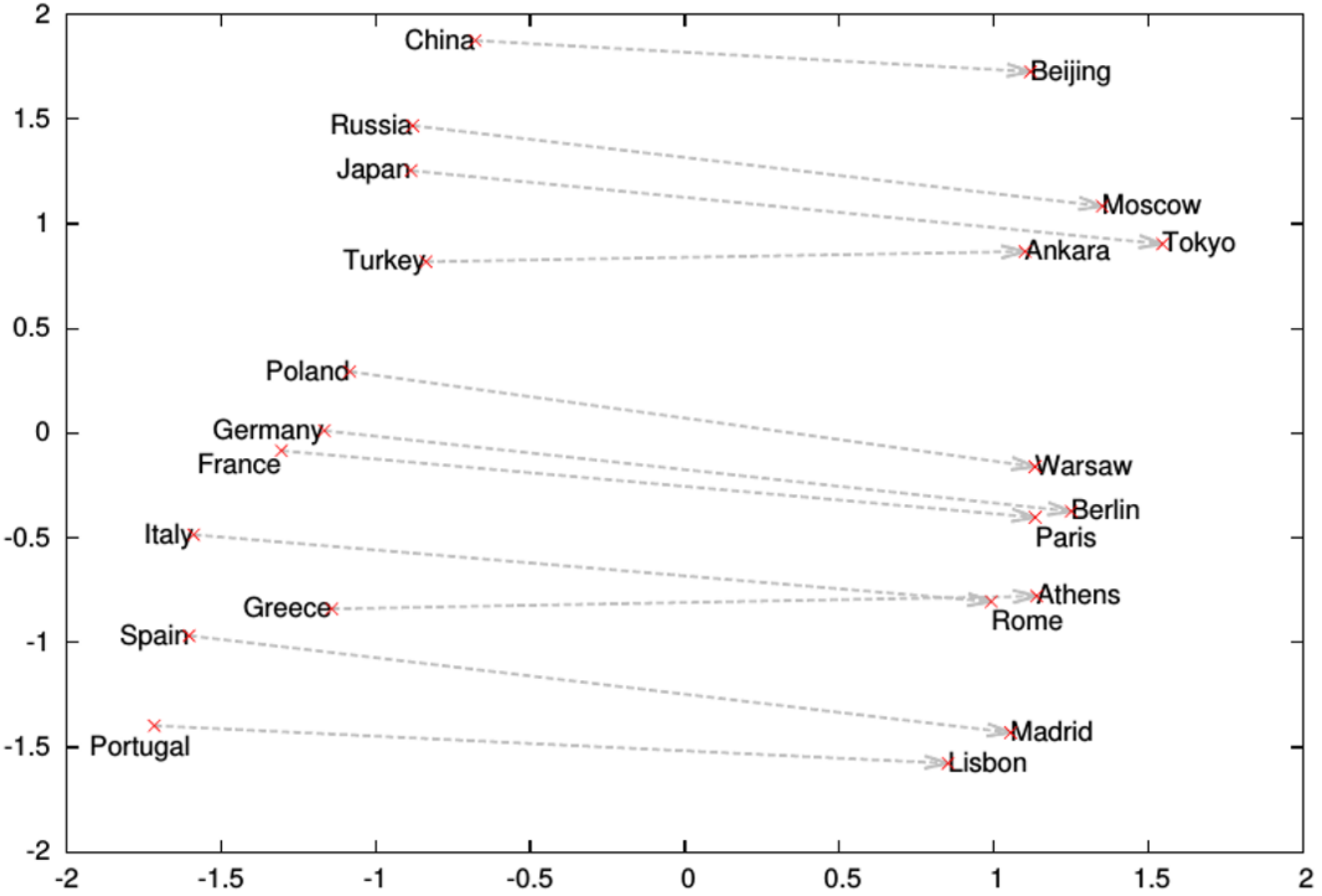
man:woman :: king:?

+	king	[0.30 0.70]
-	man	[0.20 0.20]
+	woman	[0.60 0.30]
	queen	[0.70 0.80]



www.cs.ucr.edu/~vagelis/classes/CS242/slides/word2vec.pptx

Word analogies

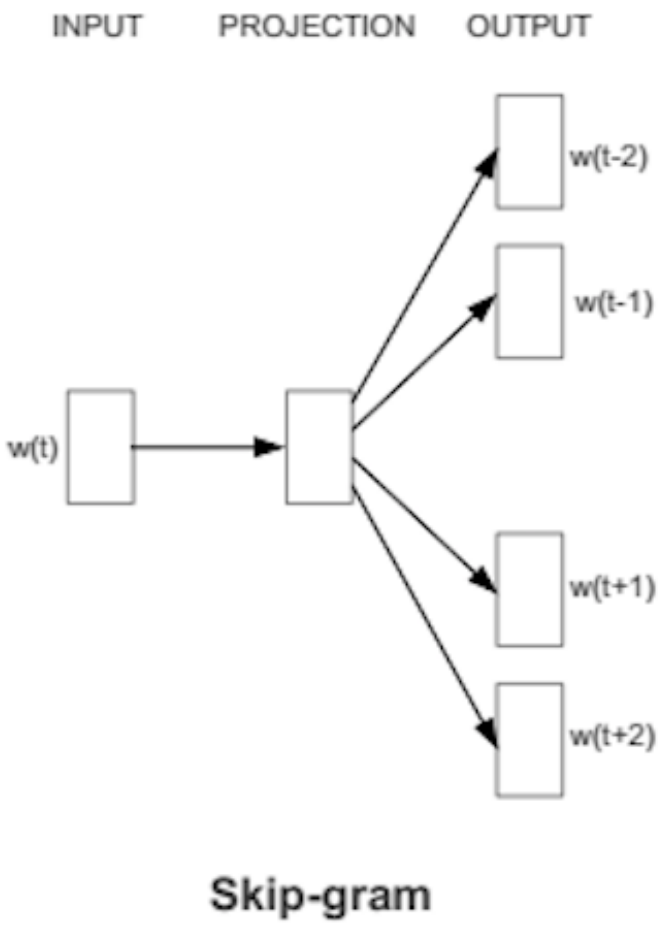


www.cs.ucr.edu/~vagelis/classes/CS242/slides/word2vec.pptx

Demo!

Skip gram

- It's a different architecture for implementing word2vec
- Key idea: **Use representation of the input word to predict the context**
  - (This is reflection of CBOW, which uses the context to predict the word)
- Start with a single word embedding and try to predict the surrounding words.
- How can this possibly work??? The embedding is what we're trying to find!

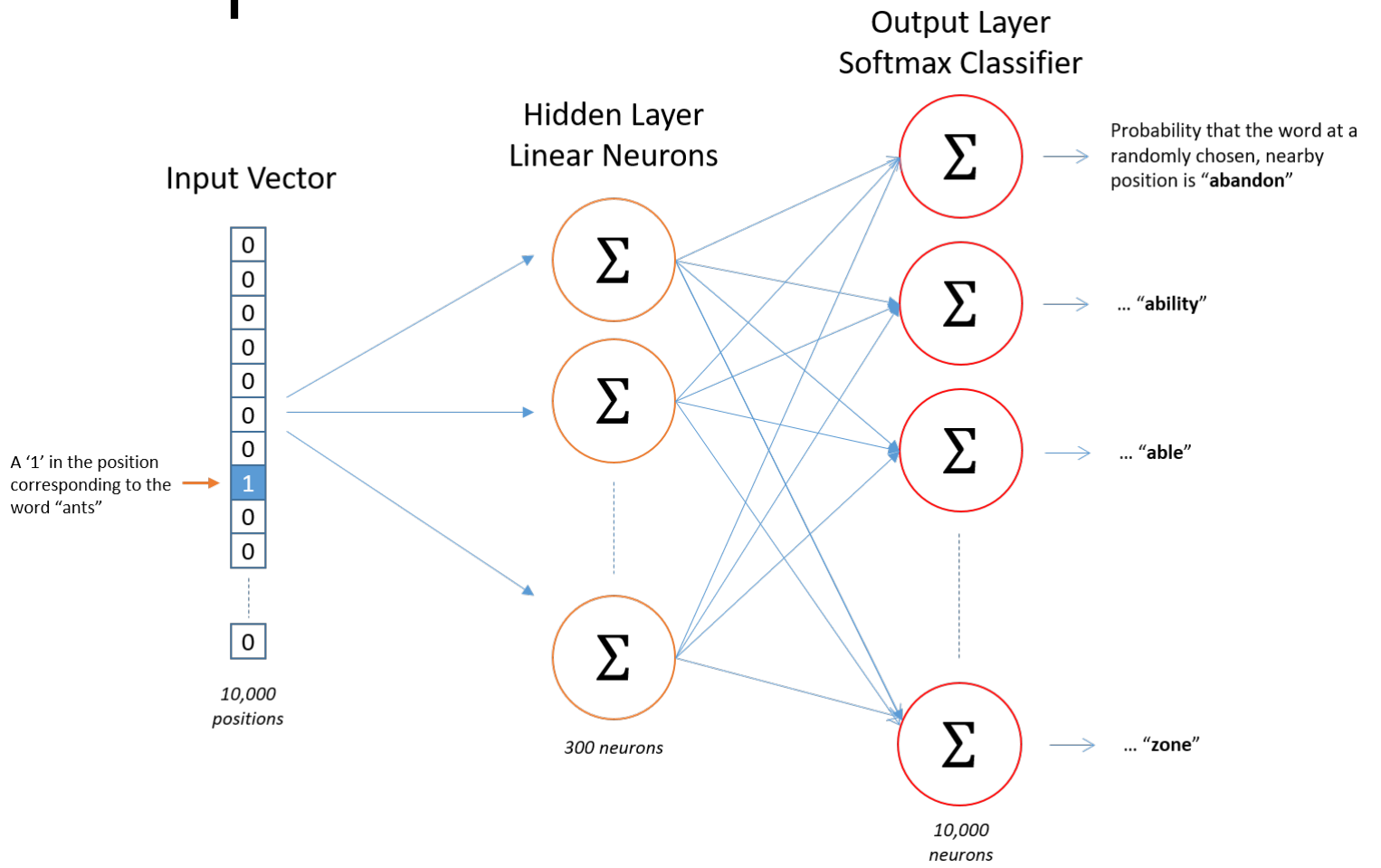


http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

Skip Gram (window 2)

First, generate a lot of target input/output pairs

Skip Gram Architecture

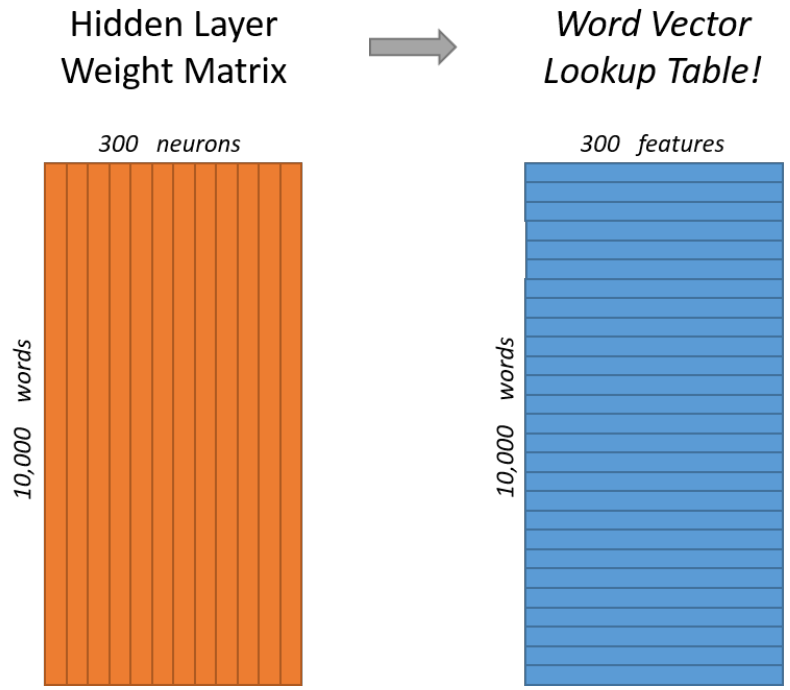


- Input is a one-hot encoding of the word. It's size 1xV.
- Hidden layer has size VxE (where E is the embedding size, a hyperparameter)
- Output layer is size 1xE, which we feed into softmax to obtain probabilities
- We have a target value for the output lauer (from previous training data step)

Skip gram example

- Vocabulary of 10,000 words.
- Embedding vectors with 300 features.
- So the hidden layer is going to be represented by a weight matrix of size 300 with 10,000 rows
- Note that if we want to emit similar results for two words that appear in similar contexts, their vectors must be similar

$$[0 \quad 0 \quad 0 \quad 1 \quad 0] \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ 10 & 12 & 19 \\ 11 & 18 & 25 \end{bmatrix} = [10 \quad 12 \quad 19]$$



http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

CBOW vs Skip gram

- CBOW needs less data, typically worse for rare words
- Skip gram needs more data, better for rare words

Word2vec shortcomings

- **Problem:** 10,000 words and 300 dim embedding gives a large parameter space to learn. And 10K words is minimal for real applications.

- Slow to train, and need lots of data, particularly to learn uncommon words.

Any ideas how to make the approach more scalable?

Word2vec improvements: word pairs and phrases

- **Idea:** Treat common word pairs or phrases as single "words."
  - E.g., Boston Globe (newspaper) is different from Boston and Globe separately. Embed Boston Globe as a single word/phrase.

- **Method:** make phrases out of words which occur together often relative to the number of individual occurrences. Prefer phrases made of infrequent words in order to avoid making phrases out of common words like "and the" or "this is".

- **Pros/cons:** Increases vocabulary size but decreases training expense.

Word2vec improvements: subsample frequent words

- **Idea:** Subsample frequent words to decrease the number of training examples.
  - The probability that we cut the word is related to the word's frequency. More common words are cut more.
  - Uncommon words (anything < 0.26% of total words) are kept
  - E.g., remove some occurrences of "the."

- **Method:** For each word, cut the word with probability related to the word's frequency.

- **Benefits:** If we have a window size of 10, and we remove a specific instance of "the" from our text:
  - As we train on the remaining words, "the" will not appear in any of their context windows.

Word2vec improvements: selective updates

- **Idea:** Use "Negative Sampling", which causes each training sample to update only a small percentage of the model's weights.

- **Observation:** A "correct output" of the network is a one-hot vector. That is, one neuron should output a 1, and *all* of the other thousands of output neurons to output a 0.

- **Method:** With negative sampling, randomly select just a small number of "negative" words (let's say 5) to update the weights for. (In this context, a "negative" word is one for which we want the network to output a 0 for). We will also still update the weights for our "positive" word.

Agenda

1. Embeddings
2. Applications and Retrieval Augmented Generation
3. Vector Databases

Vector Similarity Applications

- Clustering
- Next word prediction
- Semantic Search
- Translation

- If we can obtain a joint embedding of different input types, then we can do cross-modal similarity
- Image of a cat should map close to the word “cat”

LLMs

- Their inner workings are too much for today. For the moment, let's consider them only as systems (not as consumers of embeddings themselves)

LLMs

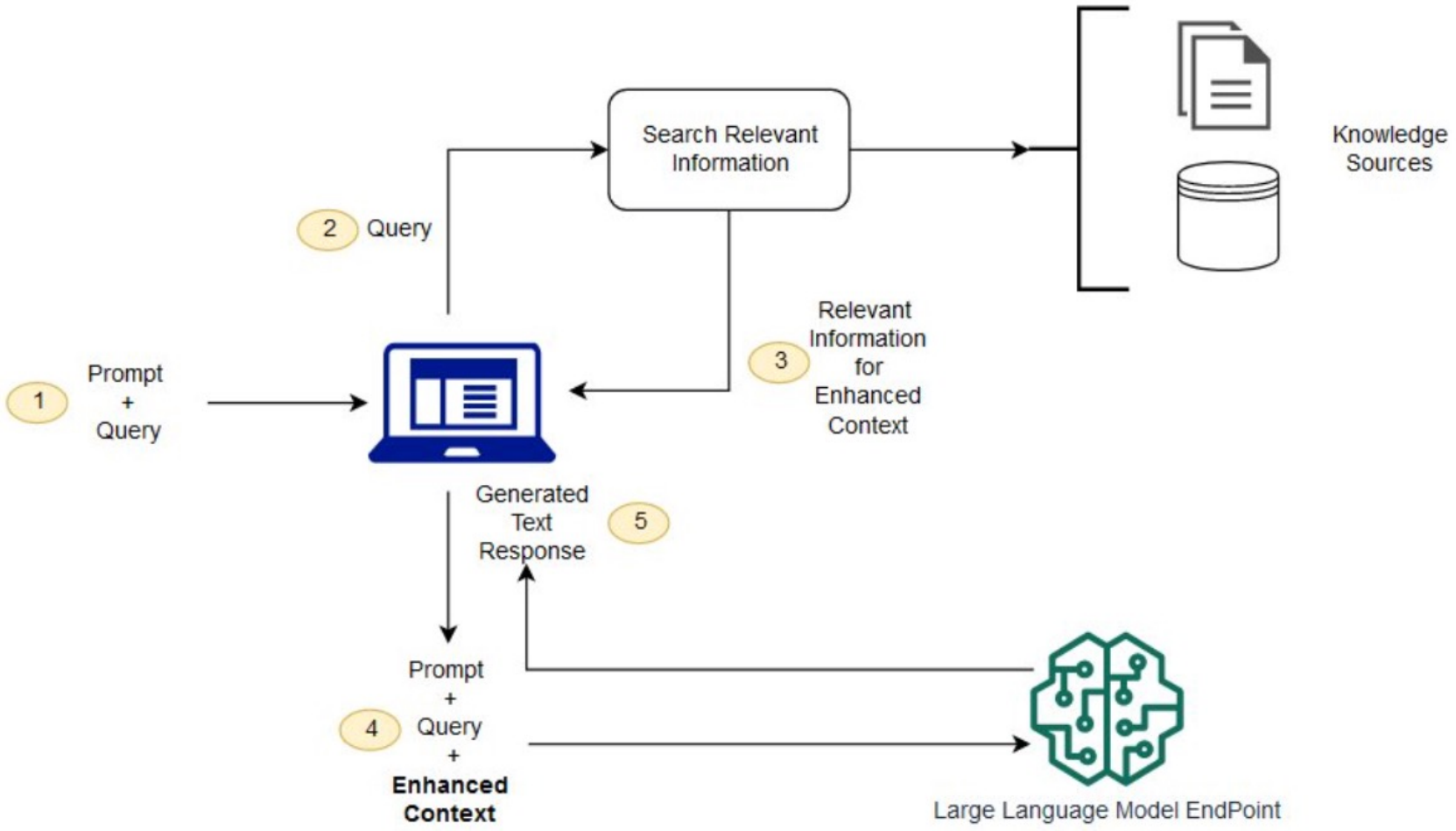
- “Write me a poem about data science”
- “Summarize the above paragraph”
- “Why is my printer not working? The answer can be found in one of the above 45,293 technical support articles”
- ”Which product is most relevant? <here are 14.3M products>”

- Today’s LLM architctures have limited max context lengths (usually 2k-32k tokens)

Retrieval Augmented Generation

- RAG "pre-builds" the LLM context

https://docs.aws.amazon.com/sagemaker/latest/dg/jumpstart-foundation-models-customize-rag.html



Retrieval Augmented Generation

- Simple, but offers lots of advantages!
- Get around the context length limit
- Improve results by editing your article database
- Avoid retraining costs
  - Expensive in general
  - Impossible if the database is changing
- Don’t have to include possibly-sensitive data in the training set
- Potentially reduces hallucinations

Other Uses

- Extended LLM "memory"
  - What if you have a ChatGPT conversation that goes very, very long?
  - The chatbot can use RAG to selectively swap in past memories from the record
- LLM answer cache
  - LLM answer generation is extremely computationally expensive
  - When a new query arrives, go back and check if we've ever answered something like that before

So What's The Problem?

```
for w in words:
  curDist = distance(q, w)
  if curDist < bestDist:
    bestDist = curDist
    bestAnswer = w
```

What is **`len(words)`**?

What if it was **`len(products)`** instead?

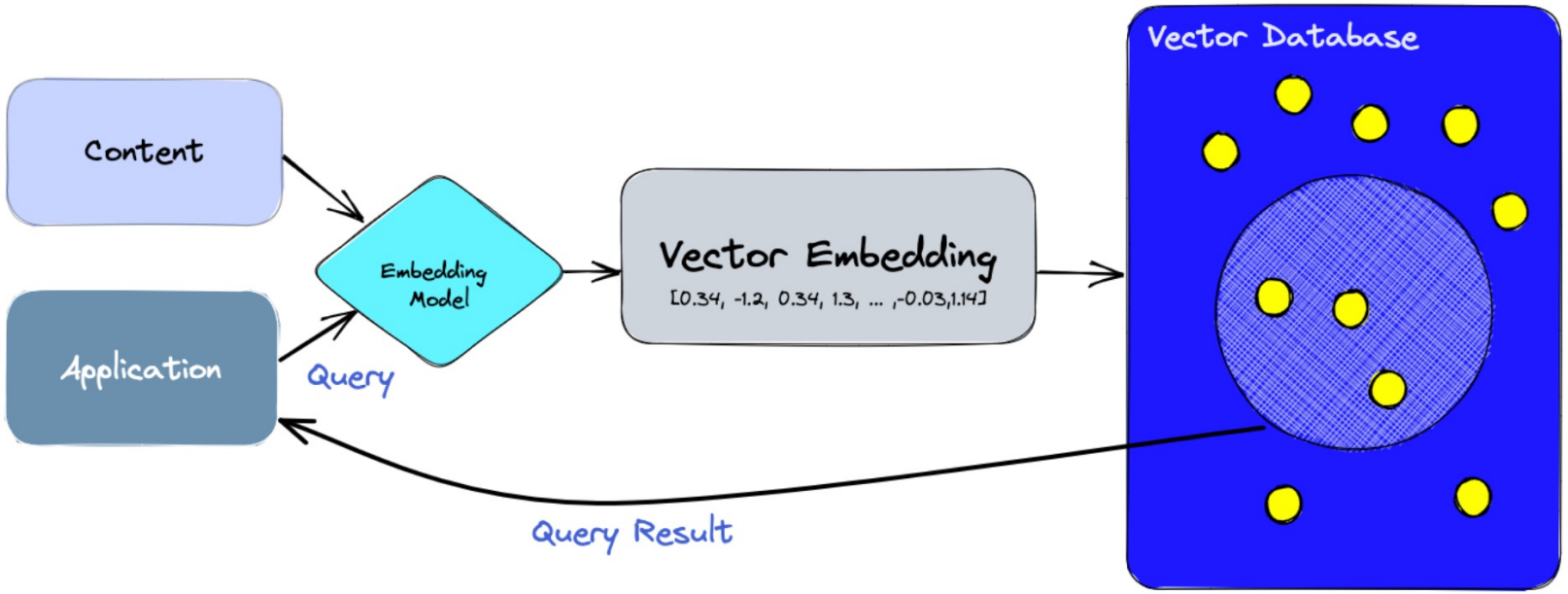
Query Efficiency

- For n points in d dimensions…
  - Linear search for nearest-neighbor is **O(nd)**
- **If d is 300, and n is vast, then this is bad**
- We need approximate nearest neighbor

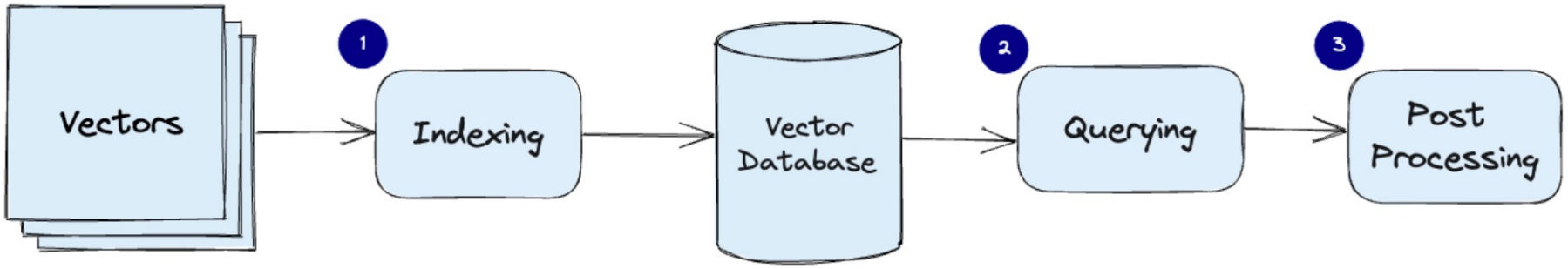
Vector Databases

- Relational databases w/VECTOR datatype
- Allows fast querying over 100Ms or even Bs of data objects
- Handles distributed operation, failure, etc

https://www.pinecone.io/learn/vector-database/



RAG Management with Vector Databases



https://www.pinecone.io/learn/vector-database/

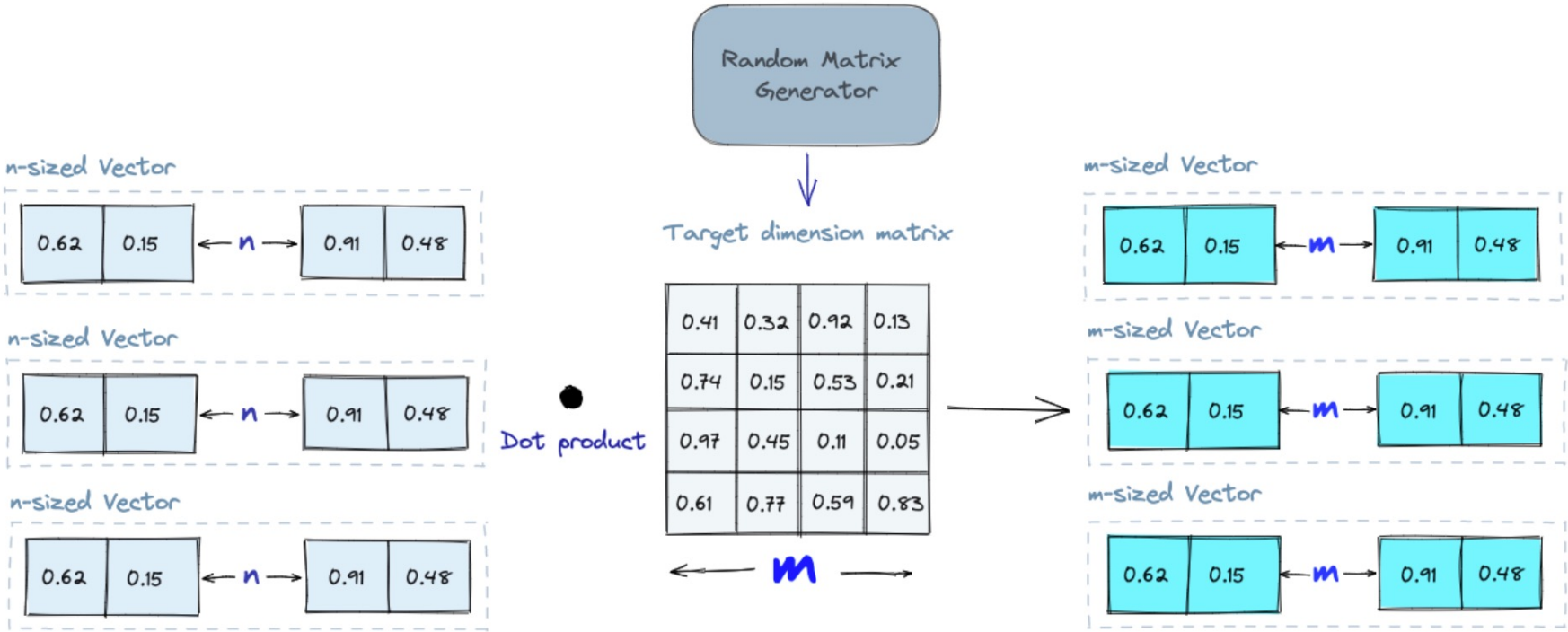
Query Efficiency

- Three basic ways to make search faster

- Vector Compression
- Hashing
- Indexing

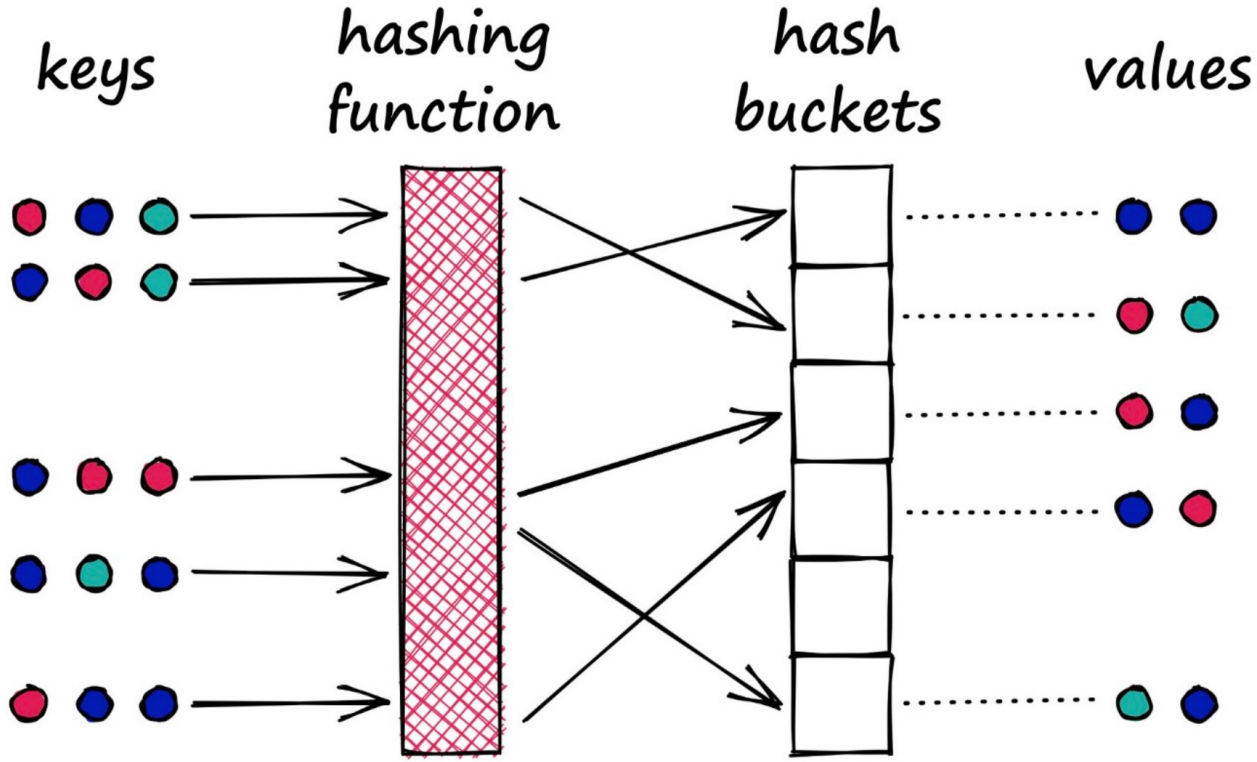
1. Vector Compression

- Reduce work by using lower-dim vectors
- **"Random projection"** is one simple method



2. Hashing

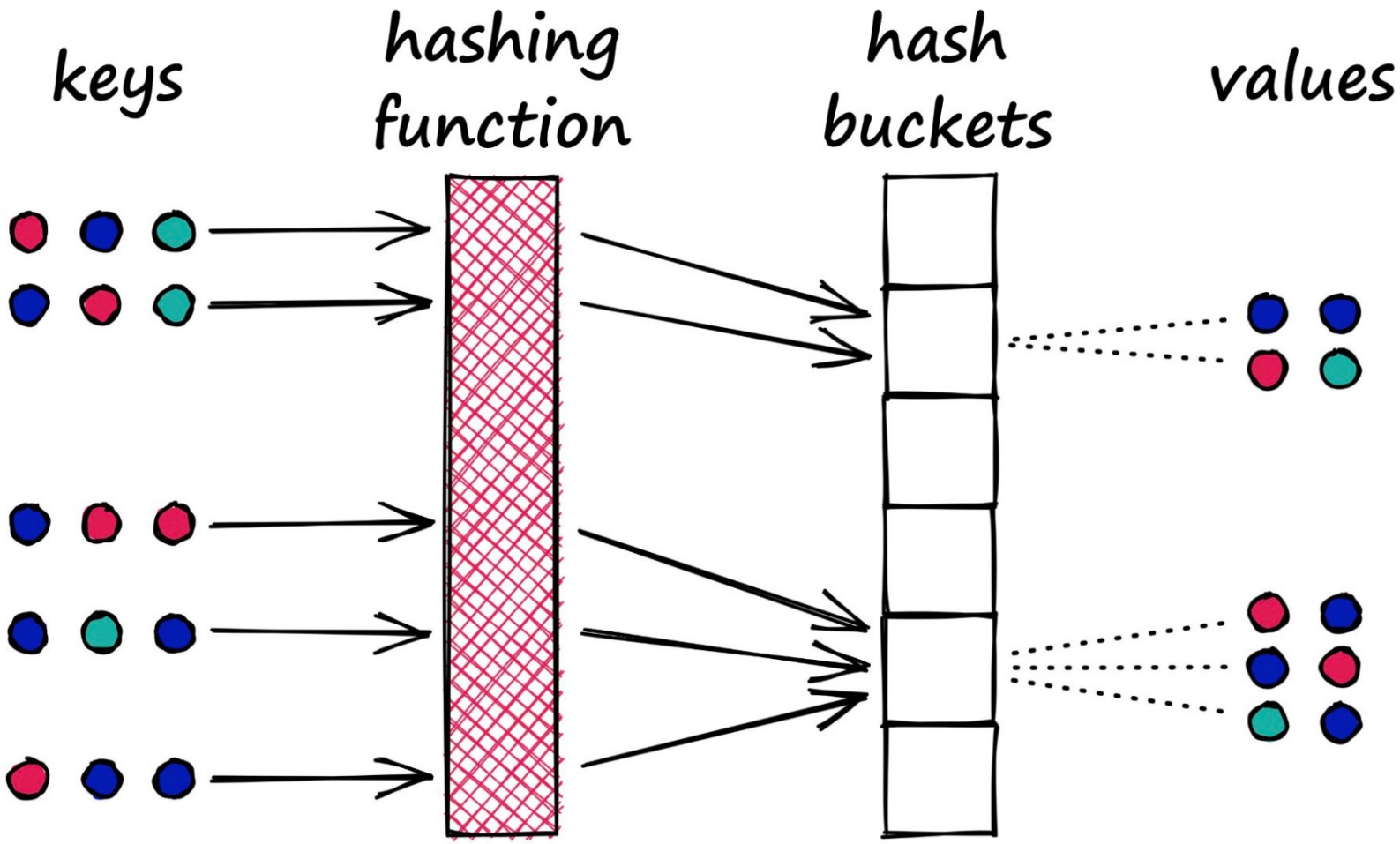
- Standard hashing gives lookups in O(1)



- Why won't they work here?
- We need **Locality Sensitive Hashing**

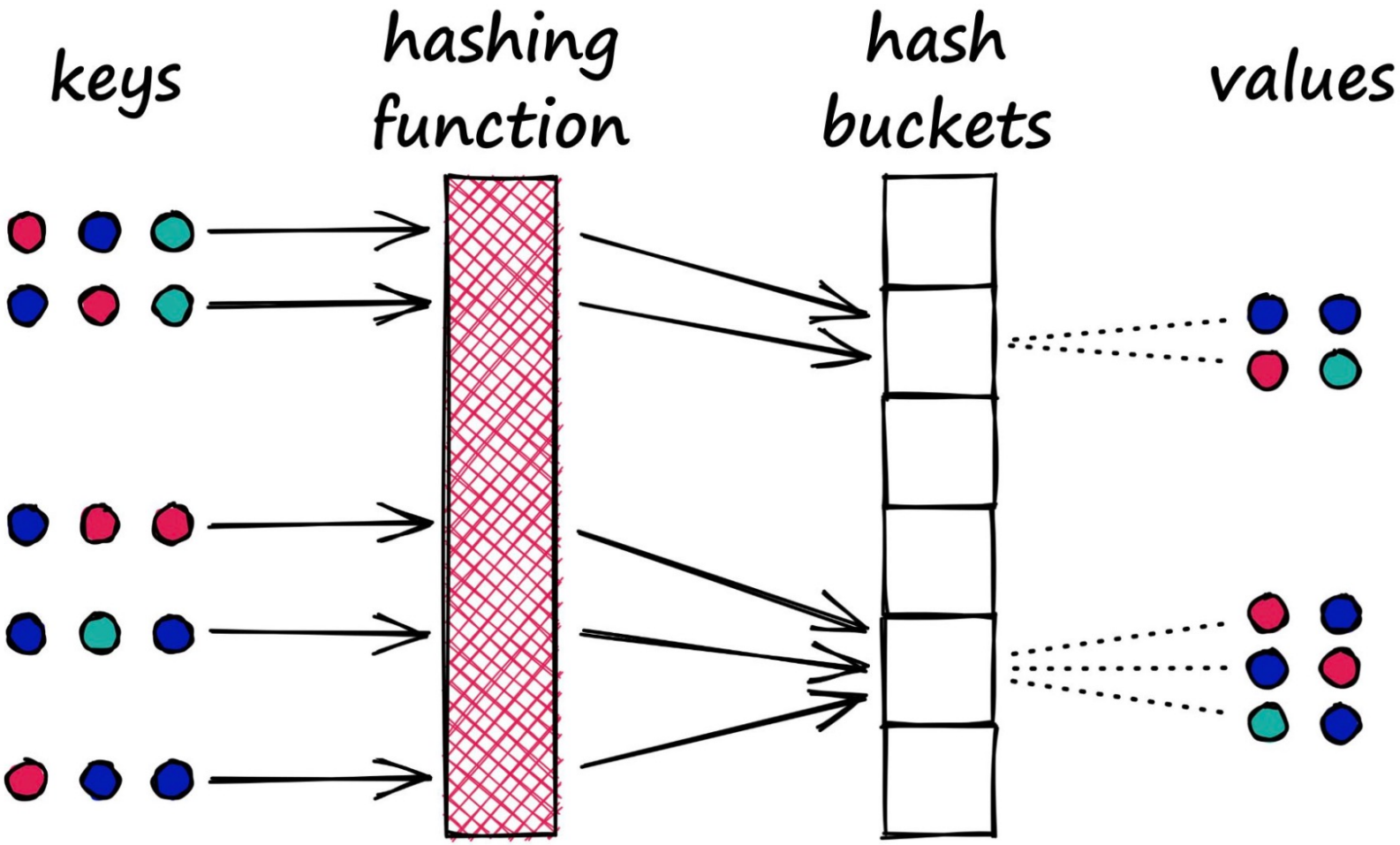
Locality Sensitive Hashing

- Traditional hashing minimizes collisions
- LSH maximizes collisions (when similar)



Locality Sensitive Hashing

- Traditional hashing minimizes collisions
- LSH maximizes collisions (when similar)

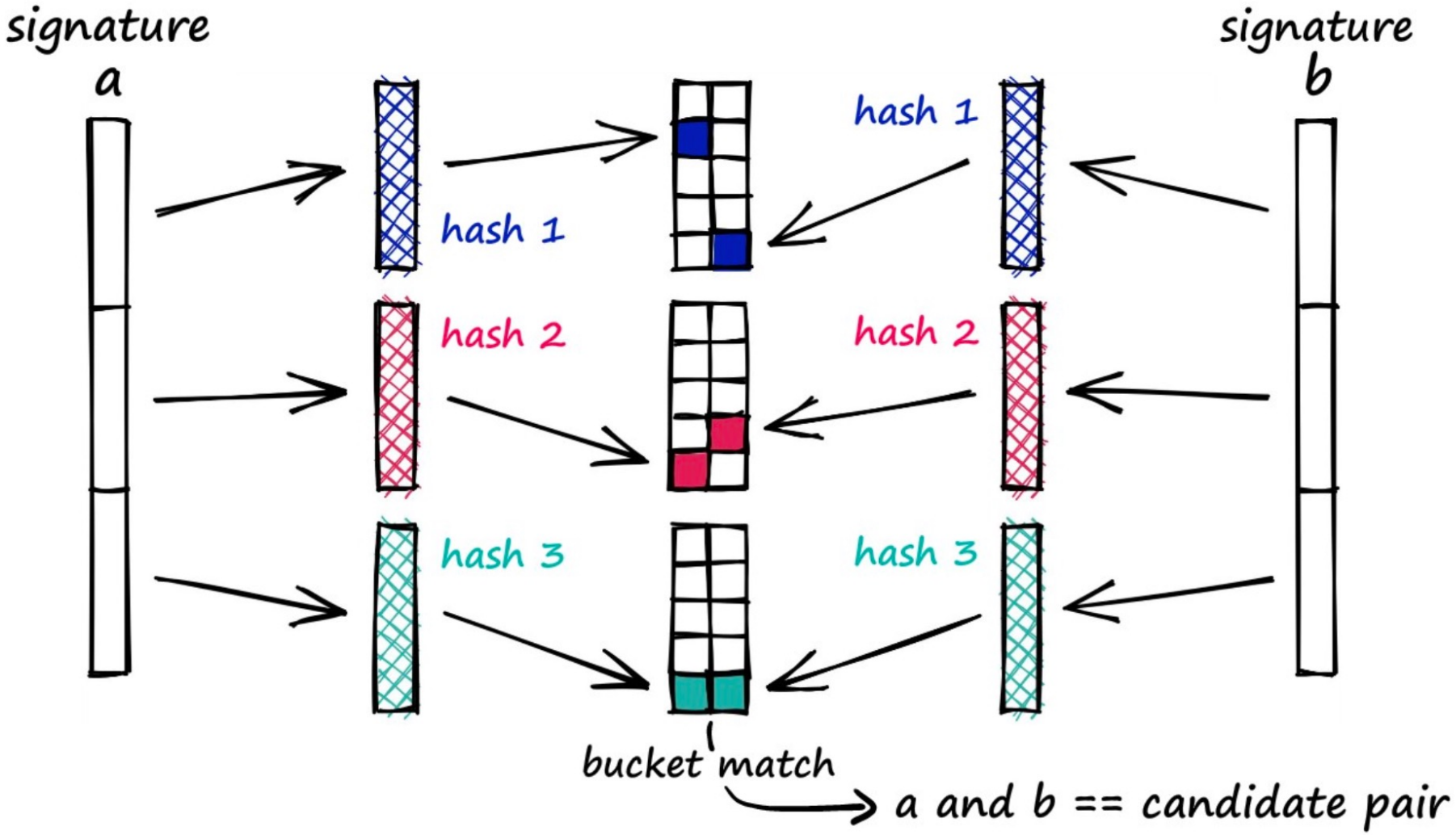


Locality Sensitive Hashing

- When inserting into database…
  - Break input v into subvectors (“signatures”)
  - Hash and store each subvector separately

- When querying database…
  - Break input q into subvectors
  - Hash each subvector separately
  - When q and some data object d share a bucket match, d is a match candidate
  - Compute traditional distance between d and all candidates

Locality Sensitive Hashing



More formally…

- Assume you have a set of hash functions where probability of bucket collision goes up when distance is smaller (aka similarity is greater)

1. Create L hash tables, each with k hash functions
2. Hash all n d-size vectors into the L tables
3. For a query q, hash it into the L tables
4. Get candidates from the tables
5. Either evaluate every candidate or terminate after "enough" items retrieved

More formally…

- Important parameters L and k
  - L is the number of tables
  - k is the "width" of hash functions associated with each l \in L

- Insert time: O(nLkt) (t is hashing time)
- Storage: O(nL)
- Larger L and k improve probability of finding a "close" point
- You can tune this probability via L, k

3. Indexing

- The same problems we saw in hashing prevent us from using conventional indexes
  - Vectors can be very similar but not identical
  - How do we order objects in 300 dimensions?

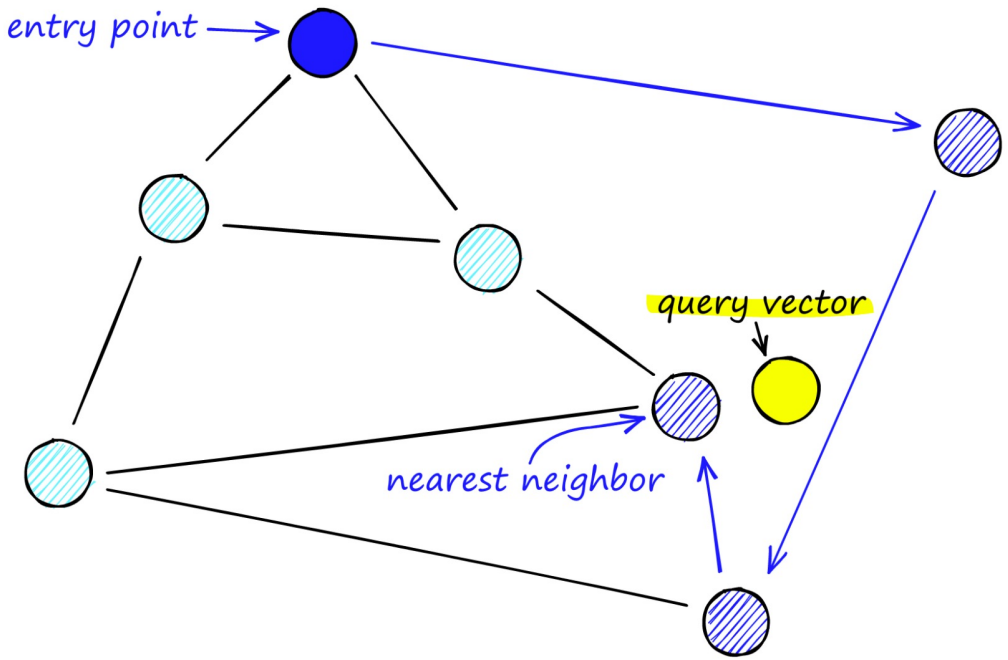
- Let's try **Navigable Small World** graph search instead, for fast approximate nearest neighbor search

This Slide Is NSW!

```
Greedy_Search(q: object, v_entry_point: object)
1  v_curr←v_entry_point;
2  δ_min←δ(q, v_curr); v_next←NIL;
3  foreach v_friend ∈ v_curr.getFriends() do
4    δ_fr←d (query, v_friend)
5    if δ_fr < δ_min then
6      δ_min←δ_fr;
7      v_next←v_friend;
8  if v_next=NIL then return v_curr;
9  else return Greedy_Search(q, v_next);
```

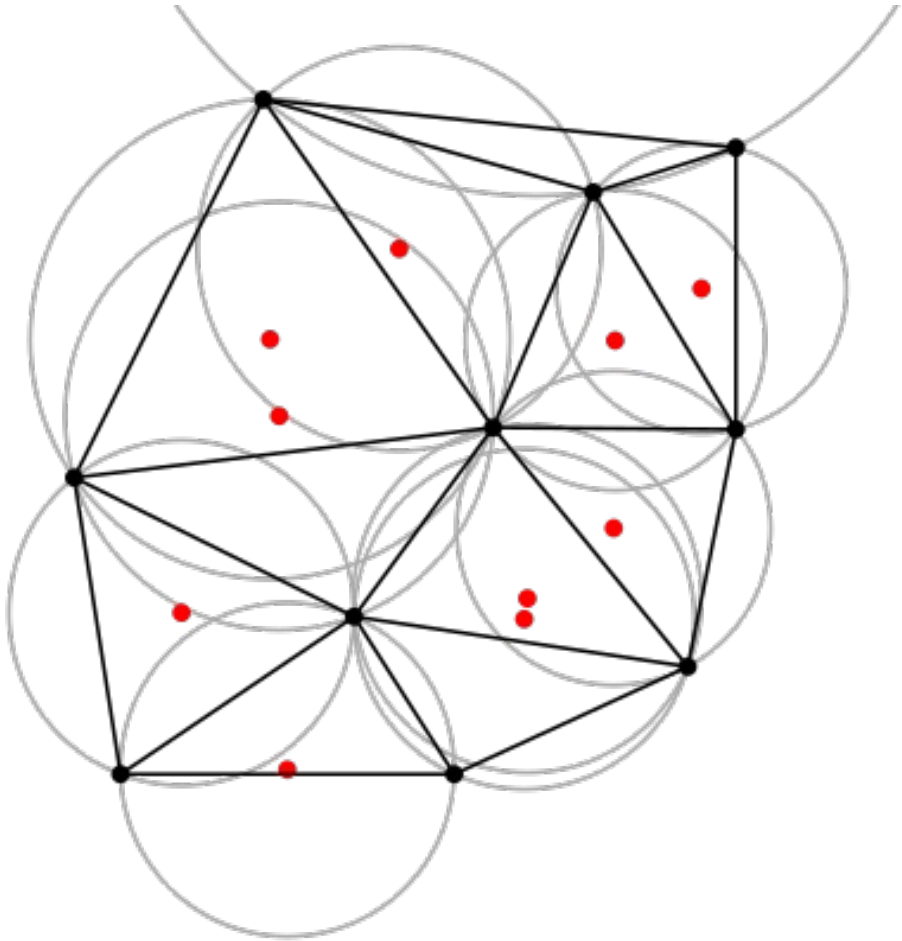
Malkov et al, "Approximate nearest neighbor algorithm based on navigable small world graphs "

Will this always work?



Delaunay Triangulation

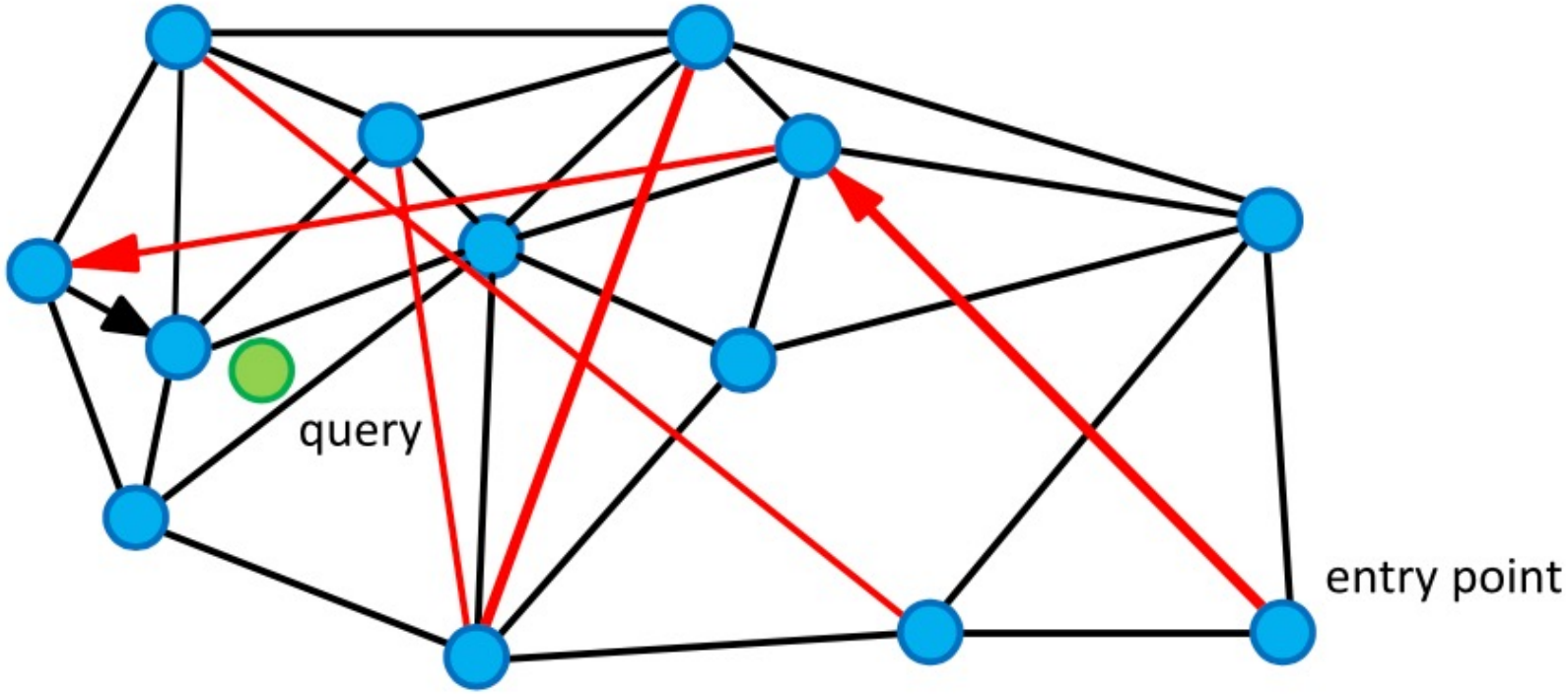
- We are only guaranteed to find the true closest node if each node's neighbors consist of its Voronoi neighbors
- If that doesn't hold, greedy search is approximate



Still, what's the problem?

NSW

- Black links are local; red are “long distance”



Malkov et al, “Approximate nearest neighbor algorithm based on navigable small world graphs “

How does this work?

- The graph is constructed s.t:
- Some links are “short-range” and approximate the Delaunay graph. This provides basic connectivity so greedy search works
- Some links are “long-range” and are used for logarithmic scaling

Malkov et al, “Approximate nearest neighbor algorithm based on navigable small world graphs “

Insertion Algorithm

- For each new incoming object:
  - Find its set of **f** closest neighbors. (Try **w** times)
  - Connect incoming object to every neighbor (and vice versa)
- **That's it! Though you have to tune f, w**

- **Key idea**: as objects are inserted, the graph is "filled-in" and old short-range links are "stretched" to become long-range ones

- **It's not perfect:** f-nearest neighbors aren't guaranteed to give us a Delaunay graph. Increasing w gives more chances to avoid greedy search failure

Malkov et al, "Approximate nearest neighbor algorithm based on navigable small world graphs "

Behavior

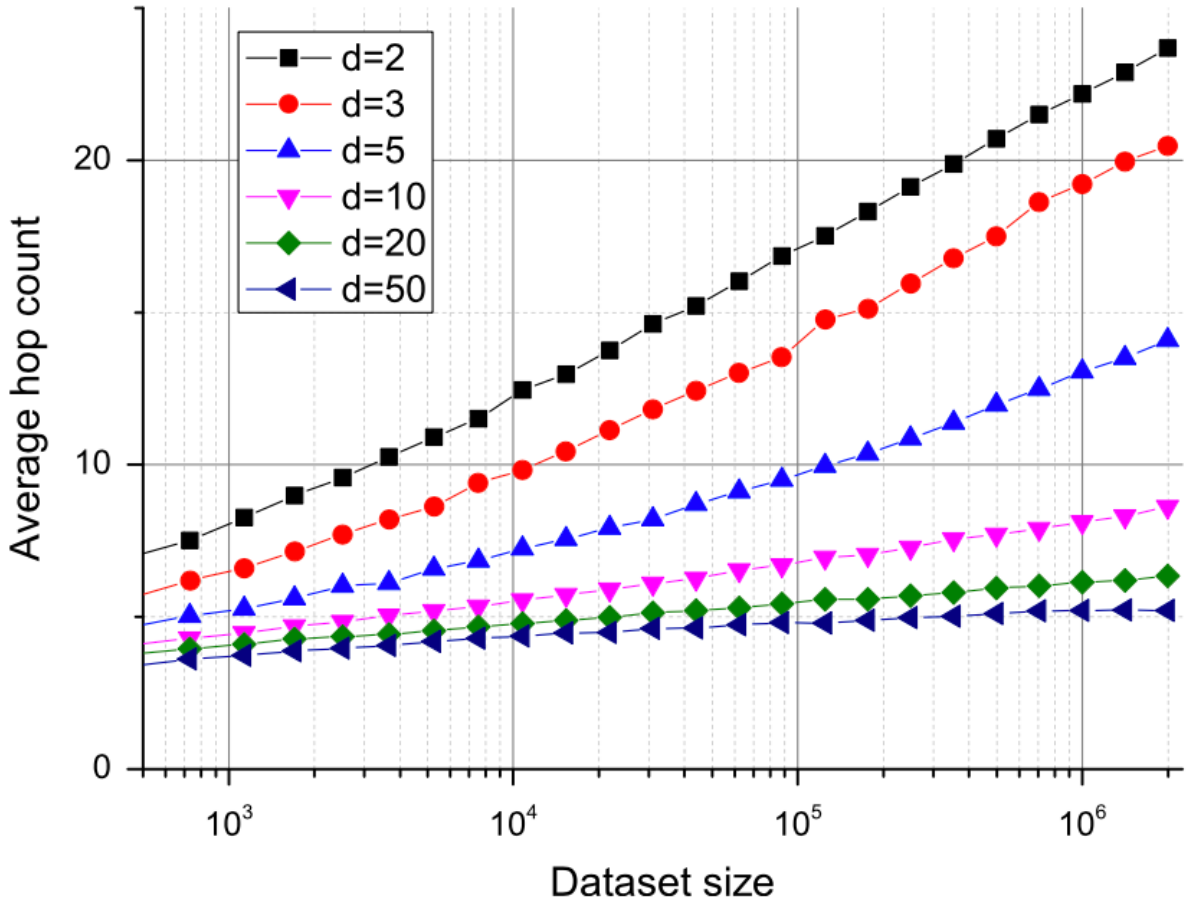


Fig. 2. The average hop count induced by a greedy search algorithm for different dimensionality Euclidean data (k=10, w=20). The navigable small world properties are evident from the logarithmic scaling.

Malkov et al, "Approximate nearest neighbor algorithm based on navigable small world graphs "

Behavior

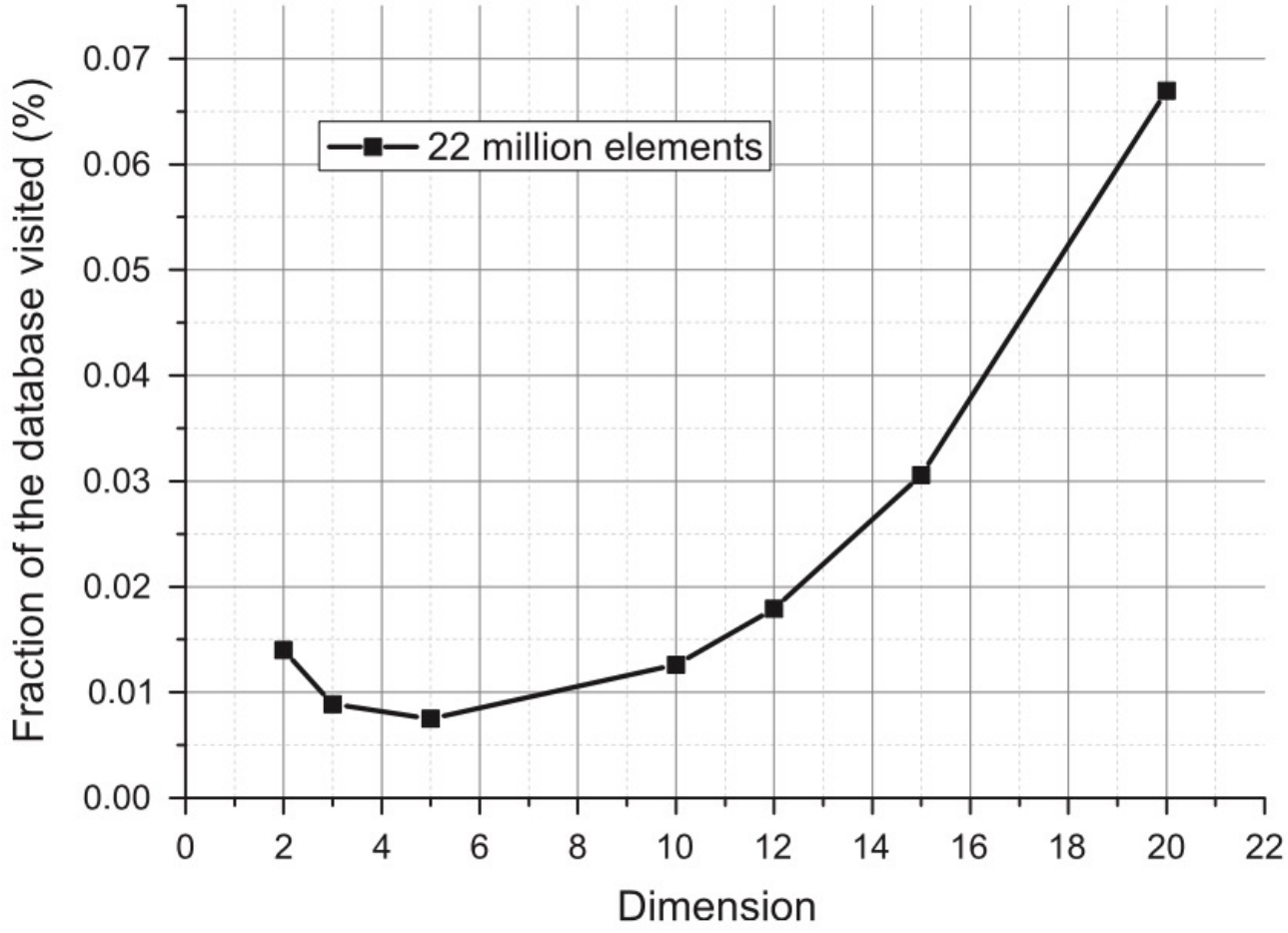
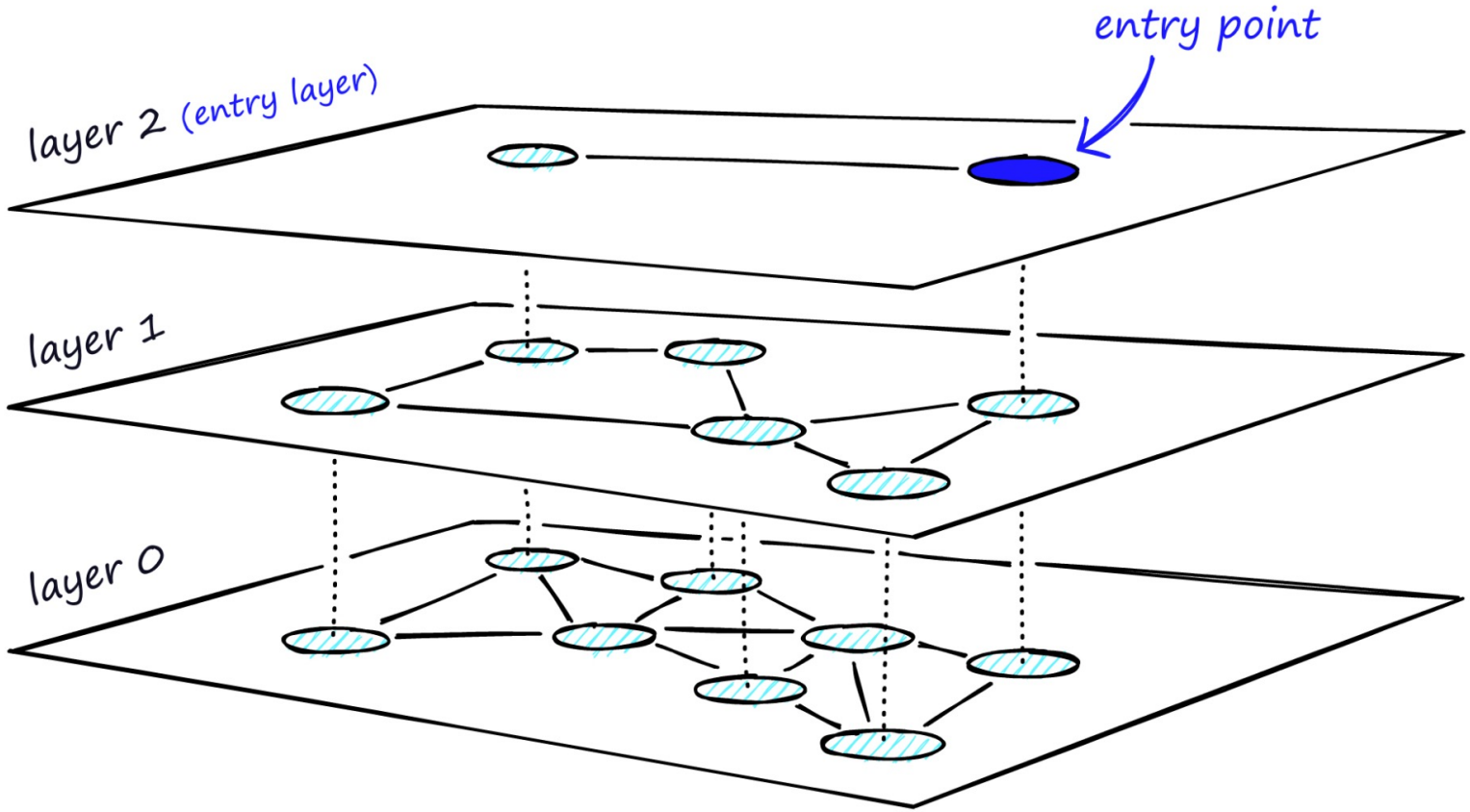


Fig. 5. Average fraction of visited elements within a single 10-NN-search with 0.999 recall for about 22 million elements dataset.

Malkov et al, "Approximate nearest neighbor algorithm based on navigable small world graphs "

HNSW

- Explicitly hierarchical versions --- where searcher transits different levels --- are also possible



https://www.pinecone.io/learn/series/faiss/hnsw/