

# 1 Tokens to Vectors: Word Embeddings

A token id like 257 is only a *label*, like a student ID number. It tells the model nothing about meaning. An **embedding** replaces the label by a list of numbers (a *vector*) chosen so that **words used in similar ways get similar vectors**. The big idea is a single sentence that powers everything from 2013’s word2vec to today’s search engines: *you can learn what something means by looking at the company it keeps*.

## 2 Tokens are just labels

In the previous notes we turned text into a list of token ids, such as [257, 260]. Can we feed those numbers straight into a neural network? Not sensibly. The number 257 is not “bigger” or “more” than 256 in any meaningful way; the ids are just names, handed out in the order the merges were learned. If a model did arithmetic on the ids it would conclude, for example, that token 260 is “close to” token 259, even if the two words mean completely unrelated things.

### 2.1 First attempt: one-hot vectors

A cleaner idea is a **one-hot vector**: a list as long as the vocabulary, with a 1 in the position of the token and 0 everywhere else (Figure 1). Now no token looks “bigger” than another. But there are two serious problems:

- **Huge and wasteful.** With 50,000 tokens, every token becomes a vector of 50,000 numbers, nearly all zeros.
- **No notion of similarity.** Every pair of different one-hot vectors is equally unrelated (they are perfectly “perpendicular”). To this representation, *cat* is exactly as different from *dog* as it is from *democracy*.

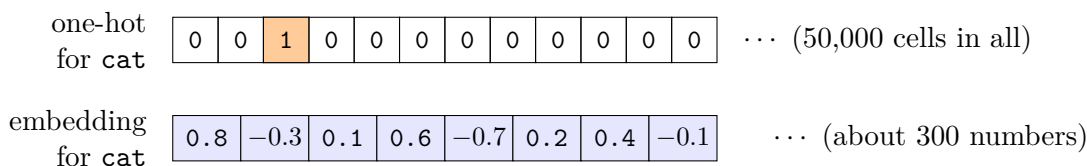


Figure 1: A one-hot vector is mostly zeros and tells us nothing about meaning. An embedding is a short list of ordinary numbers (the values shown are made up).

### 2.2 The goal: short, dense, meaningful vectors

We want something better: a vector of a few hundred numbers for each token, in which

1. **similar words have similar vectors** (*cat* and *dog* are close; *cat* and *democracy* are far apart), and
2. **directions mean something**, so that relationships between words show up as consistent differences between their vectors (more on this in Section 8.3).

This is called a **word embedding**: the word is “embedded” as a point in a space with a few hundred dimensions. The rest of these notes are about how to *get* such vectors and how to use them.

### 3 Meaning as position: the map idea

Think of a map. Two numbers (latitude and longitude) are enough to say where a city is, and cities that are close on the map are close in real life. An embedding is a *map of meanings*: instead of cities, it places words, and instead of two numbers it usually uses hundreds, because there are many more ways for meanings to be alike.

Figure 2 is a sketch with just two dimensions. Animals form one neighborhood, foods another, vehicles a third. Words in the same neighborhood have similar meanings; words in different neighborhoods do not.

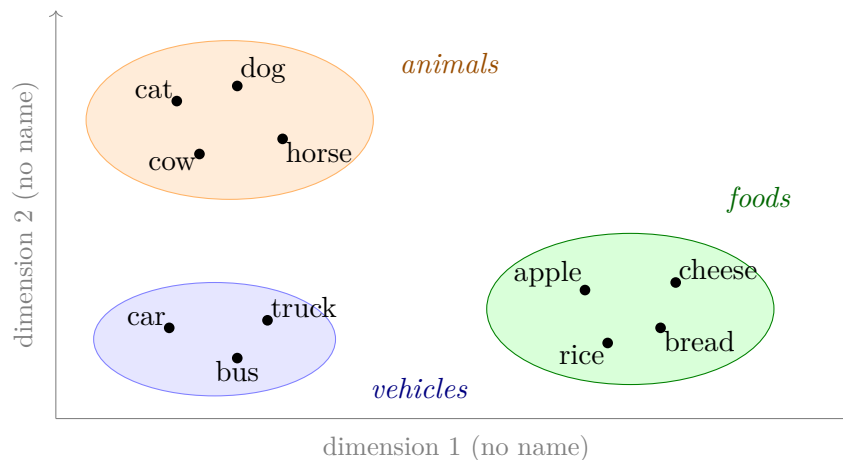


Figure 2: A made-up two-dimensional “map of meaning.” Real embeddings use hundreds of dimensions, and the dimensions do not have tidy names like “size”.

#### 3.1 Could we write the numbers by hand?

We could try. Suppose we describe each animal with four hand-chosen features, as in twenty questions: how big is it, is it furry, can it fly, does it swim?

|       | size (0-1) | furry (0/1) | flies (0/1) | swims (0/1) |
|-------|------------|-------------|-------------|-------------|
| cat   | 0.2        | 1           | 0           | 0           |
| dog   | 0.4        | 1           | 0           | 0           |
| eagle | 0.3        | 0           | 1           | 0           |
| shark | 0.9        | 0           | 0           | 1           |

Already it works a bit: the rows for *cat* and *dog* are nearly the same, and very different from *shark*. But handcrafting does not scale. Nobody can decide the right features for 50,000 tokens, for every language, and for words like *if*, *however* or *justice*. We want the computer to find good coordinates *by itself*, from examples.

## 4 Measuring closeness: dot product and cosine similarity

To say “these two vectors are similar” we need a number. The tool that is used almost everywhere in this area is **cosine similarity**, which compares the *direction* (not position) two vectors point in and ignores how long they are.

**Dot product.** For two vectors  $\mathbf{a} = (a_1, a_2, \dots)$  and  $\mathbf{b} = (b_1, b_2, \dots)$  with the same number of entries,

$$\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2 + \dots$$

Multiply matching entries and add. For example  $(2, 1) \cdot (1, 2) = 2 \cdot 1 + 1 \cdot 2 = 4$ .

**Length.** The length of a vector is  $\|\mathbf{a}\| = \sqrt{a_1^2 + a_2^2 + \dots}$ . For  $(2, 1)$  it is  $\sqrt{4 + 1} = \sqrt{5} \approx 2.24$ .

**Cosine similarity.**

$$\cos(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|}$$

This is the cosine of the angle between the two arrows. It is 1 when they point the same way, 0 when they are at right angles (“unrelated”), and  $-1$  when they point in opposite directions.

| Pair                     | Dot product                        | Cosine                       | Meaning                 |
|--------------------------|------------------------------------|------------------------------|-------------------------|
| $\mathbf{a}, \mathbf{d}$ | $2 \cdot 1 + 1 \cdot 2 = 4$        | $4/(\sqrt{5}\sqrt{5}) = 0.8$ | fairly similar          |
| $\mathbf{a}, \mathbf{c}$ | $2 \cdot (-1) + 1 \cdot 2 = 0$     | 0                            | unrelated (right angle) |
| $\mathbf{a}, \mathbf{e}$ | $2 \cdot (-2) + 1 \cdot (-1) = -5$ | $-5/(\sqrt{5}\sqrt{5}) = -1$ | opposite                |

**Closeness of meaning**  $\leftrightarrow$  **a high cosine similarity.** If all vectors are first scaled to length 1, cosine similarity equals the plain dot product, which is why many systems store “normalized” vectors and just take dot products.

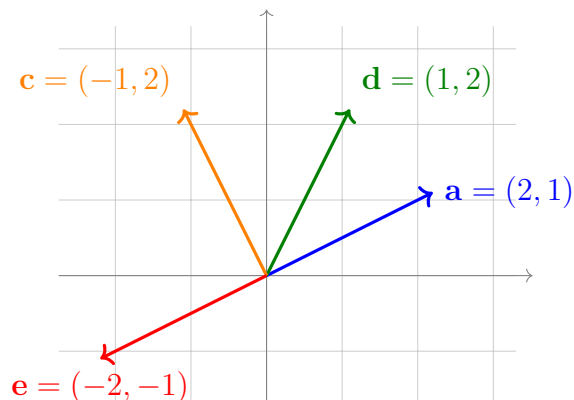


Figure 3: Four vectors in the plane. Compared with **a**: **d** points in a similar direction, **c** is at a right angle, and **e** points the opposite way.

**High dimensions are roomy.** In two dimensions there is little space. In 300 or 3000 dimensions there is a lot: two *random* vectors are almost always nearly perpendicular (cosine close to 0). That is why a model can give thousands of different words their own distinctive directions, and still have plenty of room left for related words to cluster.

## 5 The key idea: a word is known by the company it keeps

Where could a computer get meaning from? It has no eyes, no dictionary, and has never tasted an apple. What it *does* have is huge amounts of text. And text contains a powerful clue, which linguists noticed in the 1950s: **words with similar meanings are used in similar contexts.** (J. R. Firth, 1957: “You shall know a word by the company it keeps.”)

Try it yourself. What word fits in the blank?

“The \_\_\_\_\_ chased the mouse across the kitchen.”

Cat, dog, and kitten all fit. Table, democracy, and seven do not. A computer can discover that *cat* and *dog* share contexts such as “the \_\_\_\_\_ chased,” “the \_\_\_\_\_ sleeps,” and “feed the \_\_\_\_\_” simply by counting, without knowing what a cat is.

**Distributional hypothesis:** if two words keep appearing in the same surroundings, give them similar vectors. Every method in these notes, from word2vec to modern search embeddings, is a way of putting this hypothesis to work at scale.

## 6 Counting the company: co-occurrence vectors

Let us turn the hypothesis into numbers, using a deliberately tiny example. Our entire “corpus” is these six sentences:

the cat drinks milk    the dog drinks water  
the cat likes milk    the dog likes water  
the cat sleeps    the dog barks

For every word, count which words appear right next to it (within a **window** of 1 word on each side). The result is the **co-occurrence matrix**. Each row is a vector describing one word by its neighbors. It was produced by the program in Section 9.

|        | barks | cat | dog | drinks | likes | milk | sleeps | the | water |
|--------|-------|-----|-----|--------|-------|------|--------|-----|-------|
| barks  | 0     | 0   | 1   | 0      | 0     | 0    | 0      | 0   | 0     |
| cat    | 0     | 0   | 0   | 1      | 1     | 0    | 1      | 3   | 0     |
| dog    | 1     | 0   | 0   | 1      | 1     | 0    | 0      | 3   | 0     |
| drinks | 0     | 1   | 1   | 0      | 0     | 1    | 0      | 0   | 1     |
| likes  | 0     | 1   | 1   | 0      | 0     | 1    | 0      | 0   | 1     |
| milk   | 0     | 0   | 0   | 1      | 1     | 0    | 0      | 0   | 0     |
| sleeps | 0     | 1   | 0   | 0      | 0     | 0    | 0      | 0   | 0     |
| the    | 0     | 3   | 3   | 0      | 0     | 0    | 0      | 0   | 0     |
| water  | 0     | 0   | 0   | 1      | 1     | 0    | 0      | 0   | 0     |

Read row *cat*: it appeared next to *drinks* once, *likes* once, *sleeps* once, and *the* three times. Row *dog* looks almost the same, except that *sleeps* is replaced by *barks*. Now compare rows using cosine similarity (Section 4):

- $\cos(\text{cat}, \text{dog}) = \frac{9 + 1 + 1}{\sqrt{12}\sqrt{12}} = \frac{11}{12} \approx 0.917$  (very similar: they share almost all their neighbors),
- $\cos(\text{milk}, \text{water}) = 1.0$  (identical neighbors in this tiny corpus),
- $\cos(\text{cat}, \text{milk}) \approx 0.408$  (much lower: only some shared neighbors).

Nobody told the computer that cats and dogs are animals or that milk and water are liquids. The pattern in the counts revealed it.

## 6.1 Making the counts better: PMI

Raw counts have a flaw: very common words such as *the* dominate. A word that appears everywhere tells you little about any particular neighbor. **Pointwise mutual information (PMI)** fixes this by asking: *do these two words appear together more often than chance would predict?* If *drinks* and *milk* appear together much more often than their individual popularity would suggest, their PMI is high. Programs usually keep only the positive values (“PPMI”), which is what our code does.

## 6.2 Making the vectors short: SVD

The rows above have one entry per vocabulary word, so for a real vocabulary they are tens of thousands of numbers long (like one-hot vectors, but with useful information). A standard

trick from linear algebra, the **singular value decomposition (SVD)**, squeezes each long row into a short dense vector (say 300 numbers) while keeping the similarity structure as well as possible. This count-then-compress recipe was used for decades (“latent semantic analysis” dates to about 1990), and it produces perfectly usable word embeddings.

Why do *cat* and *dog* get a high similarity even though the two words never appear next to each other in our corpus? (Because their *neighbors* overlap: both appear next to *the*, *drinks*, and *likes*.)

## 7 word2vec: learning the vectors instead of counting

Count-then-compress has drawbacks: the full matrix is enormous, and every new text requires rebuilding it. In 2013 Tomas Mikolov and colleagues at Google published **word2vec**, which skips the giant table and *learns* short vectors directly by playing a simple prediction game. It was fast enough to train on billions of words, and the vectors it produced worked remarkably well. For a while it was the standard way to turn words into numbers.

### 7.1 The prediction game (skip-gram)

Pick any word in a sentence as the **center word**. The words within a fixed distance (the **window**) on either side are its **context words**. The **skip-gram** game is: *given the center word, predict which words appear around it*. (Figure 4.)

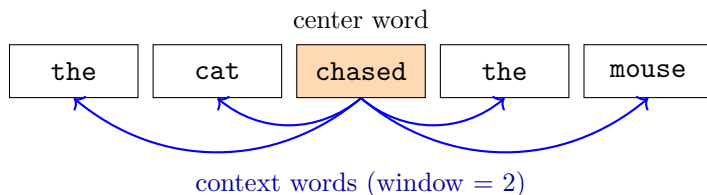


Figure 4: Skip-gram: from the center word *chased*, predict its neighbors within two words on each side.

Each sentence gives many **(center, context) training pairs**. For the sentence *the cat chased the mouse* with window 2, the center word *chased* produces four pairs:

(chased, the) (chased, cat) (chased, the) (chased, mouse)

The whole sentence gives  $2 + 3 + 4 + 3 + 2 = 14$  pairs. A big corpus gives billions.

### 7.2 What the network looks like

word2vec uses a very small neural network, shown in Figure 5. The important part is the first table, *W*. It has one row per word in the vocabulary, and each row is that word’s vector. Feeding in a one-hot vector just *selects* one row (multiplying a one-hot vector by a

table picks out a row; the program in Section 9 shows this in two lines). **That row is the embedding.** A second table,  $C$ , holds a separate “context vector” for each word, used only during training.

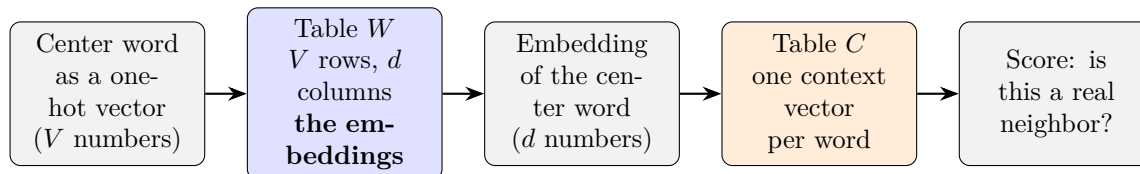


Figure 5: The word2vec network.  $V$  is the vocabulary size,  $d$  the vector length (typically 100–300). After training we keep table  $W$  and throw the rest away.

Notice how little machinery there is: no layers of complicated structure, just a table lookup, a dot product, and a score. That is why word2vec trains so quickly.

### 7.3 Negative sampling: “is this a real neighbor?”

A naive version of the game would score *every* word in the vocabulary after each step (50,000 scores, to pick the right one). That is very slow. The trick that made word2vec practical is **negative sampling**. Turn the game into many tiny yes/no questions:

1. Take a **real** pair from the text, for example (**chased**, **mouse**). The right answer is *yes*, *these are neighbors*.
2. Make up a few **fake** pairs by pairing the center word with random words from the vocabulary, for example (**chased**, **democracy**). The right answer is *no*. (Typically 5 to 20 fakes per real pair.)
3. Train the vectors so that real pairs get a high score and fake pairs get a low score.

The score of a pair is the dot product of the center word’s vector  $\mathbf{v}$  (from table  $W$ ) and the other word’s vector  $\mathbf{u}$  (from table  $C$ ). Because a dot product can be any number, we squash it into a probability between 0 and 1 using the **sigmoid** function

$$\sigma(x) = \frac{1}{1 + e^{-x}},$$

which gives  $\sigma(-4) \approx 0.02$ ,  $\sigma(0) = 0.5$ , and  $\sigma(4) \approx 0.98$ . So  $p = \sigma(\mathbf{v} \cdot \mathbf{u})$  is the model’s current guess that the pair is real.

### 7.4 One training step, by hand

Training uses a very simple rule. For each pair, let  $p = \sigma(\mathbf{v} \cdot \mathbf{u})$  and let the label be 1 for a real pair or 0 for a fake one. Then

$$\mathbf{v} \leftarrow \mathbf{v} - \eta(p - \text{label}) \mathbf{u},$$

where  $\eta$  (“eta”) is a small number called the *learning rate*. (The vector  $\mathbf{u}$  is nudged in the same spirit.) Read the rule as a story:

- For a **real** pair,  $p$  – label is negative, so we *add* a bit of  $\mathbf{u}$  to  $\mathbf{v}$ : the two vectors move **together**.
- For a **fake** pair,  $p$  – label is positive, so we *subtract* a bit of  $\mathbf{u}$  from  $\mathbf{v}$ : the vectors move **apart**.

This is the “pull together, push apart” pattern that repeats in these notes.

**Numbers.** Let  $\eta = 0.5$ , the center vector  $\mathbf{v} = (1, 0)$ , the real neighbor  $\mathbf{u}_+ = (0, 1)$ , and one random fake word  $\mathbf{u}_- = (1, -1)$ .

- Real pair:  $\mathbf{v} \cdot \mathbf{u}_+ = 0$ , so  $p = \sigma(0) = 0.5$  and  $p - 1 = -0.5$ .
- Fake pair:  $\mathbf{v} \cdot \mathbf{u}_- = 1$ , so  $p = \sigma(1) \approx 0.731$  and  $p - 0 = 0.731$ .
- Update:  $\mathbf{v}' = (1, 0) - 0.5[(-0.5)(0, 1) + 0.731(1, -1)] \approx (0.635, 0.616)$ .

After one step, the dot product with the real neighbor rose from 0 to about 0.62, and the dot product with the fake word fell from 1 to about 0.02. (Figure 6.) Repeat this billions of times and the vectors arrange themselves.

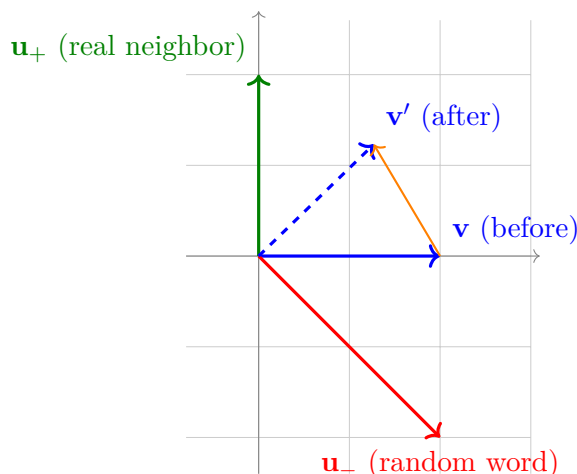


Figure 6: One training step. The center vector  $\mathbf{v}$  is pulled toward the real neighbor  $\mathbf{u}_+$  and pushed away from the random word  $\mathbf{u}_-$ , moving to  $\mathbf{v}'$ .

## 7.5 Why does this produce meaning?

Consider *cat* and *dog*. Both keep appearing next to *chased*, *sleeps*, and *the*. Each time, they are both pulled toward the same context vectors. Words that are pulled toward the same places end up close to each other. Meanwhile *cat* and *democracy* are pulled in different directions and drift apart. In other words, **word2vec compresses the co-occurrence**

statistics of Section 6 into short vectors, without ever building the giant table. (Researchers later proved a precise version of this: skip-gram with negative sampling is closely related to factorizing a shifted PMI matrix. The two routes lead to nearly the same place.)

## 7.6 A second version: CBOW

word2vec comes in two flavors. **Skip-gram** predicts the context words from the center word. **CBOW** (continuous bag of words) does the reverse: it averages the vectors of the context words and predicts the center word. CBOW is a little faster; skip-gram tends to do better on rare words. Everything above applies to both.

## 7.7 Practical details

- Typical settings: vectors of 100 to 300 numbers, a window of about 5, and 5 to 20 negative samples per real pair.
- **Subsampling.** Very frequent words (*the*, *of*) are randomly skipped during training, since they carry little information.
- **How negatives are chosen.** Fake words are drawn according to  $(\text{frequency})^{0.75}$ , which picks rare words somewhat more often than their raw frequency would.
- **Famous release.** Google published vectors trained on about 100 billion words of news text: 3 million words and phrases, 300 numbers each.

# 8 What word2vec learns

## 8.1 Seeing it work on a toy world

The code in Section 9 trains word2vec on 300 random sentences about animals, foods and places, such as *the dog runs in the park*. After 30 passes over the data it knows nothing except which words keep each other's company. Yet:

| Word  | Nearest words (cosine)                    |             |
|-------|-------------------------------------------|-------------|
| cat   | horse (0.99), dog (0.97), cow (0.96)      | all animals |
| apple | cheese (0.99), bread (0.98), rice (0.97)  | all foods   |
| park  | store (1.00), school (0.99), house (0.99) | all places  |

Also  $\cos(\text{cat}, \text{dog}) = 0.97$  but  $\cos(\text{cat}, \text{apple}) = 0.42$ . Figure 7 flattens the 8-dimensional vectors to 2 dimensions so that we can look at them: the three categories form three separate neighborhoods that the program discovered on its own.



Figure 7: The word2vec vectors from our toy program, reduced from 8 dimensions to 2 (with a method called PCA). The three kinds of words end up in three separate neighborhoods. The axes themselves have no meaning.

## 8.2 Watching it learn

The program also records how wrong the model is after each pass over the data (the **loss**). Figure 8 shows the real numbers. The loss drops quickly at first, as the vectors find a sensible arrangement, then flattens out. It does not go to zero, because the fake pairs are chosen at random, so some guesses are always a bit uncertain.

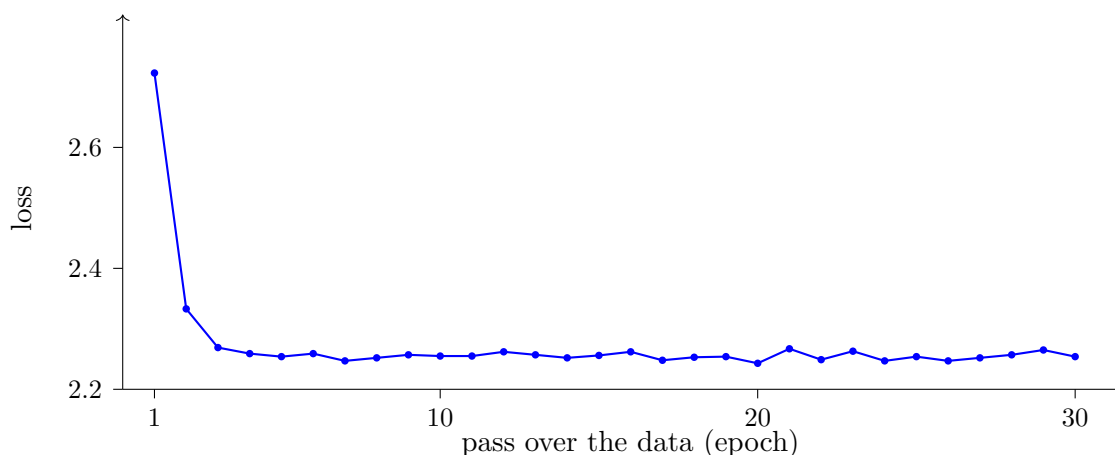


Figure 8: Training loss of the toy word2vec model, one point per pass over the data. (The vertical axis starts at 2.2, not at 0, to make the flattening visible.)

## 8.3 The famous analogies

On large corpora, word2vec showed an unexpected bonus: *directions* in the space capture relationships. The classic example:

$$\mathbf{v}(\text{king}) - \mathbf{v}(\text{man}) + \mathbf{v}(\text{woman}) \approx \mathbf{v}(\text{queen}).$$

Reading it as geometry (Figure 9): the step from *man* to *woman* is roughly the same arrow as the step from *king* to *queen*. The same trick finds things like capital cities (*Paris* – *France* + *Italy*  $\approx$  *Rome*). It happens because the context patterns of “king” and “queen” differ in the same way as those of “man” and “woman”.

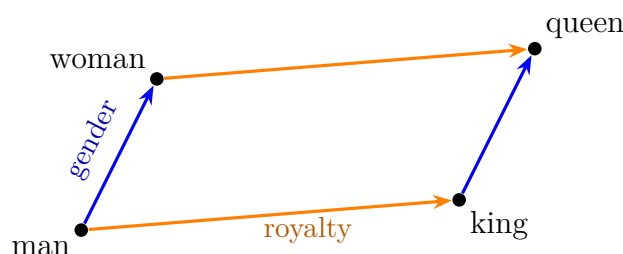


Figure 9: A hand-drawn sketch of the analogy idea in two dimensions. Related pairs differ by (roughly) the same arrow, so a gender step and a royalty step can be added like building blocks.

## 8.4 A relative: GloVe

In 2014 a Stanford team published **GloVe**, which comes at the same goal from the count-based side: it fits vectors so that the dot product of two word vectors matches the logarithm of how often the words co-occur in the whole corpus. In practice GloVe and word2vec vectors are similar in quality. GloVe is a useful reminder that “count” and “predict” methods are two views of the same idea.

# 9 The Python code

Below is a complete program in plain Python with one library, **numpy** (for arrays of numbers). It contains everything from the previous sections: cosine similarity, co-occurrence counting, PPMI with SVD, and a small but real word2vec trainer.

## 9.1 Part 1: small helpers

`one_hot` builds a one-hot vector, `cosine` is the formula from Section 4, and `most_similar` lists the words closest to a given word.

```
"""Turning words into vectors, in plain numpy (written to be read by
students)."""

import numpy as np

# -----
# Part 1: small helpers
# -----
```

```

def build_vocab(sentences):
    """Return the sorted list of words and a dict word -> id."""
    words = sorted({w for s in sentences for w in s.split()})
    return words, {w: i for i, w in enumerate(words)}

def one_hot(i, size):
    """A vector of zeros with a single 1 in position i."""
    v = np.zeros(size)
    v[i] = 1.0
    return v

def cosine(a, b):
    """Cosine similarity: 1 = same direction, 0 = unrelated, -1 = opposite
    ."""
    return float(a @ b / (np.linalg.norm(a) * np.linalg.norm(b)))

def most_similar(word, words, word2id, vectors, top=3):
    """The `top` words whose vectors point most nearly the same way."""
    v = vectors[word2id[word]]
    scores = [(cosine(v, vectors[i]), w) for i, w in enumerate(words) if w
               != word]
    return sorted(scores, reverse=True)[:top]

```

## 9.2 Part 2: count-based vectors

This is Section 6 as code. `neighbor_pairs` finds every (center, neighbor) pair within the window. `cooccurrence_matrix` counts them, `ppmi` turns counts into “more together than chance” scores, and `svd_embeddings` squeezes the long rows into short vectors.

```

# -----
# Part 2: count-based vectors ("the company a word keeps")
# -----
def neighbor_pairs(sentences, word2id, window):
    """All (center, neighbor) id pairs with the neighbor within `window`
    words."""
    pairs = []
    for s in sentences:
        ids = [word2id[w] for w in s.split()]
        for pos, center in enumerate(ids):
            lo = max(0, pos - window)
            hi = min(len(ids), pos + window + 1)
            for other in range(lo, hi):
                if other != pos:
                    pairs.append((center, ids[other]))
    return pairs

def cooccurrence_matrix(sentences, word2id, window=1):
    """counts[i, j] = how often word j appears near word i."""

```

```

n = len(word2id)
counts = np.zeros((n, n))
for center, other in neighbor_pairs(sentences, word2id, window):
    counts[center, other] += 1
return counts

def ppmi(counts):
    """Positive pointwise mutual information: 'more together than by
    chance?'"""
    total = counts.sum()
    row = counts.sum(axis=1, keepdims=True)
    col = counts.sum(axis=0, keepdims=True)
    with np.errstate(divide="ignore", invalid="ignore"):
        pmi = np.log(counts * total / (row * col))
    pmi[~np.isfinite(pmi)] = 0.0
    return np.maximum(pmi, 0.0)

def svd_embeddings(matrix, dim):
    """Squeeze each long row of the matrix down to `dim` numbers."""
    u, s, _ = np.linalg.svd(matrix)
    return u[:, :dim] * np.sqrt(s[:dim])

```

### 9.3 Part 3: word2vec (skip-gram with negative sampling)

This is Section 7. Inside the loop, find these four lines, which are the heart of the algorithm: pick random fake words (`negs`), score the real and fake pairs (`p`), measure how wrong each guess is (`g`), and nudge the vectors (`W[center] -= ...`). That last line is the update rule from Section 7.4. Table `W` holds the embeddings and `C` the context vectors.

```

# -----
# Part 3: word2vec (skip-gram with negative sampling)
# -----
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

def train_sgns(sentences, dim=8, window=2, negatives=5, epochs=30,
               lr=0.05, seed=0):
    """Learn one vector per word. Returns words, word2id, vectors, losses.
    """
    rng = np.random.default_rng(seed)
    words, word2id = build_vocab(sentences)
    n = len(words)
    pairs = np.array(neighbor_pairs(sentences, word2id, window))

    # Noise distribution for picking "wrong" words: frequency ** 0.75
    freq = np.bincount(pairs[:, 1], minlength=n).astype(float)
    noise = freq ** 0.75
    noise /= noise.sum()

```

```

W = (rng.random((n, dim)) - 0.5) / dim    # word vectors (the
      embeddings)
C = np.zeros((n, dim))                    # context vectors (helpers)
losses = []

for epoch in range(epochs):
    rng.shuffle(pairs)
    total = 0.0
    for center, context in pairs:
        negs = rng.choice(n, size=negatives, p=noise)
        ids = np.concatenate(([context], negs))
        labels = np.zeros(len(ids))
        labels[0] = 1.0                    # first one is the real
            neighbor
        v = W[center].copy()
        U = C[ids]
        p = sigmoid(U @ v)                 # predicted "is a real
            neighbor?"
        total -= np.log(p[0]) + np.sum(np.log(1.0 - p[1:]))
        g = p - labels                     # how wrong each prediction
            is
        W[center] -= lr * (g @ U)           # nudge the word vector
        np.subtract.at(C, ids, lr * np.outer(g, v)) # nudge the
            contexts
    losses.append(total / len(pairs))
return words, word2id, W, losses

```

## 9.4 Part 4: a toy world to learn from

Real word2vec needs a huge corpus. To keep our demonstration small and fast, this part generates 300 random sentences from five templates about animals, foods, and places. `pca_2d` flattens vectors to two numbers each so that we can draw them (Figure 7).

```

# -----
# Part 4: a toy world to learn from
# -----
def make_toy_corpus(n_sentences=300, seed=1):
    """Random sentences about animals, foods and places."""
    rng = np.random.default_rng(seed)
    animals = ["cat", "dog", "horse", "cow"]
    foods = ["apple", "bread", "rice", "cheese"]
    places = ["school", "park", "store", "house"]
    templates = [
        "the {A} sleeps in the {P}",
        "the {A} runs in the {P}",
        "the {A} eats the {F}",
        "we cook the {F} in the {P}",
        "we buy the {F} at the {P}",
    ]
    sentences = []
    for _ in range(n_sentences):

```

```

        t = templates[rng.integers(len(templates))]
        t = t.replace("{A}", animals[rng.integers(4)])
        t = t.replace("{F}", foods[rng.integers(4)])
        t = t.replace("{P}", places[rng.integers(4)])
        sentences.append(t)
    return sentences

def pca_2d(vectors):
    """Squash vectors down to 2 numbers each, so we can draw them."""
    centered = vectors - vectors.mean(axis=0)
    _, _, vt = np.linalg.svd(centered, full_matrices=False)
    return centered @ vt[:2].T

```

## 9.5 Part 5: a demonstration

```

# -----
# Demo
# -----
if __name__ == "__main__":
    # 1. An embedding table is just a lookup (and a one-hot product)
    table = np.array([[0.1, 0.9], [0.8, 0.2], [0.5, 0.5]])
    print("row 1 by lookup :", table[1])
    print("row 1 by one-hot:", one_hot(1, 3) @ table)

    # 2. Counting neighbors in a tiny corpus
    tiny = ["the cat drinks milk", "the dog drinks water",
            "the cat likes milk", "the dog likes water",
            "the cat sleeps", "the dog barks"]
    words, word2id = build_vocab(tiny)
    counts = cooccurrence_matrix(tiny, word2id, window=1)
    print()
    print("columns:", words)
    for w, row in zip(words, counts):
        print(f"{w:7s}", row.astype(int))
    cat, dog = counts[word2id["cat"]], counts[word2id["dog"]]
    milk, water = counts[word2id["milk"]], counts[word2id["water"]]
    print("cos(cat, dog)      =", round(cosine(cat, dog), 3))
    print("cos(milk, water)   =", round(cosine(milk, water), 3))
    print("cos(cat, milk)     =", round(cosine(cat, milk), 3))

    # 3. word2vec on the toy world
    corpus = make_toy_corpus()
    print()
    print("example sentences:", corpus[:3])
    words, word2id, W, losses = train_sgns(corpus)
    print("loss at epochs 1, 10, 20, 30:",
          [round(float(losses[i]), 3) for i in (0, 9, 19, 29)])
    for w in ["cat", "apple", "park"]:
        near = most_similar(w, words, word2id, W, top=3)
        print(f"nearest to {w:5s}:", [(n, round(s, 2)) for s, n in near])

```

```

print("cos(cat, dog)   =", round(cosine(W[word2id["cat"]], W[word2id["
    dog"]])), 2))
print("cos(cat, apple) =", round(cosine(W[word2id["cat"]], W[word2id["
    apple"]])), 2))

# 4. The count-based route (PPMI + SVD) on the same corpus
big_counts = cooccurrence_matrix(corpus, word2id, window=2)
P = svd_embeddings(ppmi(big_counts), dim=4)
near = most_similar("cat", words, word2id, P, top=3)
print("count-based nearest to cat:", [(n, round(s, 2)) for s, n in
    near])

```

## 9.6 What the program prints

```

row 1 by lookup : [0.8 0.2]
row 1 by one-hot: [0.8 0.2]

columns: ['barks', 'cat', 'dog', 'drinks', 'likes', 'milk', 'sleeps', 'the',
    ', 'water']
barks    [0 0 1 0 0 0 0 0 0]
cat      [0 0 0 1 1 0 1 3 0]
dog      [1 0 0 1 1 0 0 3 0]
drinks   [0 1 1 0 0 1 0 0 1]
likes    [0 1 1 0 0 1 0 0 1]
milk     [0 0 0 1 1 0 0 0 0]
sleeps   [0 1 0 0 0 0 0 0 0]
the      [0 3 3 0 0 0 0 0 0]
water    [0 0 0 1 1 0 0 0 0]
cos(cat, dog)      = 0.917
cos(milk, water)   = 1.0
cos(cat, milk)     = 0.408

example sentences: ['the horse eats the cheese', 'the cat sleeps in the
    house', 'the dog runs in the park']
loss at epochs 1, 10, 20, 30: [2.723, 2.255, 2.243, 2.254]
nearest to cat : [('horse', 0.99), ('dog', 0.97), ('cow', 0.96)]
nearest to apple: [('cheese', 0.99), ('bread', 0.98), ('rice', 0.97)]
nearest to park : [('store', 1.0), ('school', 0.99), ('house', 0.99)]
cos(cat, dog)     = 0.97
cos(cat, apple)   = 0.42
count-based nearest to cat: [('horse', 1.0), ('dog', 1.0), ('cow', 1.0)]

```

Things to notice:

- The first two lines show that an “embedding lookup” and “multiply by a one-hot vector” give the same answer. This is the whole trick behind every embedding layer in every neural network.
- The co-occurrence table matches Section 6.
- The word2vec route and the count-based route find the same neighbors of *cat*. They

are two roads to the same place.

In `train_sgns`, which line makes two vectors move *closer*, and which makes them move *apart*? (Both come from the same line, `W[center] -= lr * (g @ U)`. For the real neighbor the label is 1, so `g` is negative and the line adds that context vector. For the random words the label is 0, so `g` is positive and the line subtracts them.)

## 10 Where word2vec falls short

word2vec was a breakthrough, but it has well-known limits. Knowing them explains everything that came afterward.

1. **One vector per word, no matter the meaning.** The word *bank* gets a single vector, a blend of “river bank” and “money bank.” Words like *bat*, *light*, and *play* have the same problem. This is called **polysemy**.
2. **Unknown words.** A word that was not in the training text has no vector at all. Typos, new slang, names, and rare word forms (*unfriendliest*) are all missing. (Compare the “unknown token” problem from the tokenization notes.)
3. **It only knows words, not sentences.** How do you get one vector for “cheap flights to Paris”? Averaging the word vectors throws away word order (“dog bites man” = “man bites dog”).
4. **It absorbs the biases in its data.** The vectors learn from text written by people, so they pick up stereotypes. Studies found, for instance, that some occupations ended up closer to “man” or “woman” than they should be. Biased embeddings can silently pass those biases to any system built on them.
5. **Fixed after training.** The vectors do not change with the topic of the document or with what is being asked.

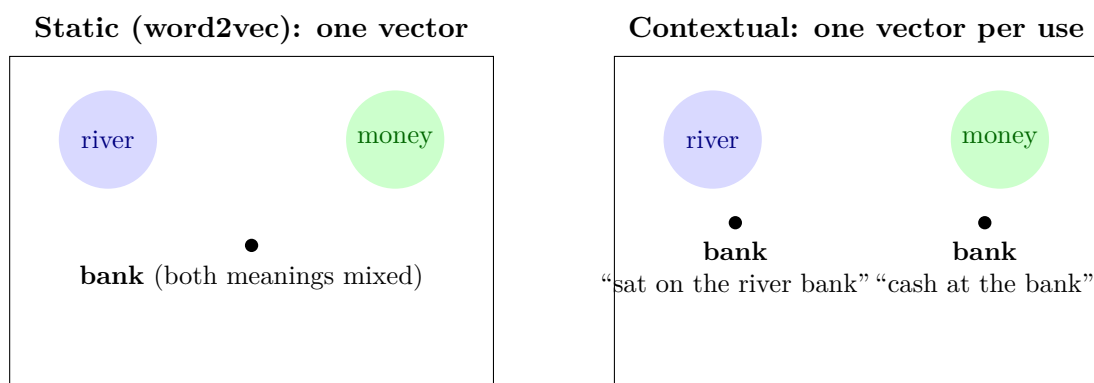


Figure 10: The polysemy problem. word2vec gives *bank* one vector sitting between its two meanings. A contextual model gives each *use* of the word its own vector.

## 11 Upgrades: subwords and context

### 11.1 fastText: use the pieces of words

**fastText** (2016) keeps the word2vec game but builds each word’s vector from the vectors of its character pieces (for example the three-to-six-letter chunks of *unfriendlyest*). A brand-new word is then assembled from pieces it has seen before, so unknown words are no longer a problem. This is the same idea as subword tokenization from the previous notes: when the whole word is unfamiliar, fall back to its parts.

### 11.2 Contextual embeddings: the vector depends on the sentence

The big fix for polysemy arrived in 2018 with **ELMo** and **BERT**. Instead of looking a word up in a fixed table and stopping, these models run the whole sentence through a neural network (a transformer, in BERT’s case) and read off a vector for each token *after* the network has looked at all the surrounding words. Now *bank* in “river bank” and *bank* in “bank loan” get different vectors (Figure 10, right panel).

The picture to keep in mind:

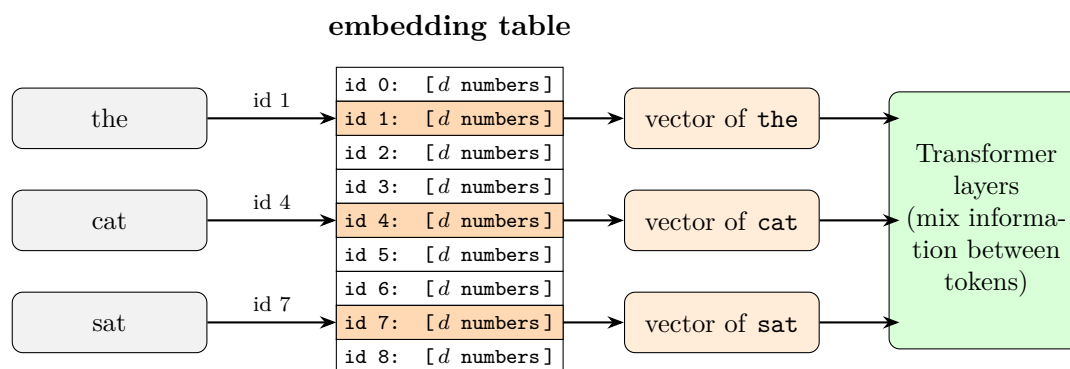


Figure 11: The first step of every transformer is an embedding lookup, just like word2vec’s table  $W$ . The transformer layers that follow then update each vector using the surrounding tokens.

### 11.3 The embedding layer of a language model

This gives a clear way to think about modern language models. Their first step is exactly a word2vec-style **embedding table**, with one row per *token id* (Figure 11), turned into a first vector for each token. The difference is how it is trained. Instead of the neighbor-prediction game, the table is learned *together with the whole network*, by training the model to predict the next token. The rows of the table adjust to be whatever helps that task, and in practice they end up capturing similarity much like word2vec does. Size example: 100,000 tokens with 4,096 numbers each is about 410 million numbers, and this table is only the front door of the model.

The word “embedding” is used for two related things. (1) The *input embedding table* inside a language model: a lookup from token id to a starting vector, one fixed vector per token. (2) An *embedding model* (the topic of the next sections): a whole network whose job is to output *one vector for an entire sentence or document*, built to be compared by cosine similarity. Chatbots use (1); search and recommendation systems mostly use (2).

## 12 From words to sentences: embeddings for text

Most real tasks need vectors for *whole pieces of text*: a search query, a paragraph, a product review, an email. We want

$$\text{text} \longrightarrow \text{one vector } \mathbf{t}$$

such that two texts with similar *meaning* get similar vectors, even if they share no words: “How do I fix a flat tire?” should land near “repairing a punctured bicycle wheel.”

**Naive approach: average the word vectors.** Simple, fast, surprisingly decent as a baseline, but it ignores word order and context.

**Better: Sentence-BERT (2019).** Take a contextual model such as BERT and *train it for the job*: feed in a whole sentence, combine the output vectors (for example by averaging them) into one vector, and train so that sentences with similar meaning get similar vectors. How do we train for that? With the pull-together, push-apart idea once more.

### 12.1 Contrastive learning: pull together, push apart

Collect millions of **pairs** of texts that belong together: a question and a page that answers it, a title and its article, two paraphrases of one sentence, or an image and its caption. Then train in *batches*. Within one batch of  $N$  pairs, each text’s partner is the **positive**, and the other  $N - 1$  partners in the batch are **negatives** for free. We compute the similarity of every question with every answer, producing an  $N \times N$  table (Figure 12), and train the model so that in each row the correct partner has the highest score.

|            | answer 1             | answer 2             | answer 3             |
|------------|----------------------|----------------------|----------------------|
| question 1 | <b>pull together</b> | push apart           | push apart           |
| question 2 | push apart           | <b>pull together</b> | push apart           |
| question 3 | push apart           | push apart           | <b>pull together</b> |

Figure 12: One training batch of three (question, answer) pairs. The diagonal holds the true matches, which should get the highest similarity. The other six cells are free negatives. Real batches hold hundreds or thousands of pairs.

For the curious, the standard loss for one question  $q$  with true answer  $a^+$  in a batch of answers  $a_1, \dots, a_N$  is

$$-\log \frac{e^{s(q, a^+)/\tau}}{\sum_{j=1}^N e^{s(q, a_j)/\tau}},$$

where  $s$  is cosine similarity and  $\tau$  is a “temperature” setting. It is the loss that asks “out of all the answers in this batch, how sure are you of the right one?” If you squint, this is the same pull-and-push game as word2vec’s negative sampling, with sentences instead of words and a whole batch of negatives instead of five random ones.

Whether the items are words, sentences, documents, or images, the recipe is the same: **(1)** collect pairs that belong together, **(2)** treat everything else as a non-match, **(3)** train a network so that matching pairs get a high cosine similarity and non-matches get a low one.

## 13 The state of the art

**How good is word2vec compared to today’s methods?** For many tasks, modern *text-embedding models* are much stronger: they understand word order, resolve word meaning from context, handle hundreds of languages, accept paragraphs or even whole documents, and can be steered with instructions. word2vec still has a place (it is tiny, fast, runs on a laptop, and is perfect for teaching), but when people today say they need “embeddings” for search or AI applications, they almost always mean the models described below.

**The state of the art in one paragraph (October 2026).** The best text-embedding models are no longer small, purpose-built networks. They are *large language models retrained to output a vector*, using contrastive learning on huge numbers of text pairs, synthetic training data written by other language models, and carefully chosen hard negatives. They follow instructions, cover many languages, and offer shortened (Matryoshka) vectors. The newest ones place text, images, audio and video in one shared space, and small open models of under a billion parameters, such as Google’s EmbeddingGemma 2, now do this too.

### 13.1 Recipe of a modern text-embedding model

Most strong embedding models released in recent years share a recipe, which is a scaled-up version of Section 12.1:

1. **Start from a large pre-trained language model.** Instead of a small network trained from scratch, take a big transformer that already understands language (a BERT-style model, or increasingly a decoder-style large language model with billions of parameters) and adapt it to output one vector per text. Qwen3-Embedding, NV-Embed, and the E5-Mistral line are examples built on language models, and Google’s Gemini Embedding models are built from Gemini.
2. **Stage 1: contrastive pre-training on huge numbers of pairs.** Hundreds of millions of weakly matched text pairs scraped from the web (titles and bodies, questions and answers, ...).

3. **Stage 2: fine-tuning on carefully curated data**, including *hard negatives*: wrong answers that look tempting, which teaches fine distinctions.
4. **Synthetic data.** Use a strong language model to *write* extra training examples (queries, documents, variations) for different tasks and languages. Several recent models rely on this.
5. **Instructions.** Many models let you prefix a short task description (“Given a web search query, retrieve relevant passages”) so that one model can behave differently for search, classification, or clustering.

## 13.2 Matryoshka embeddings: one vector, many sizes

Long vectors cost memory and slow down search. **Matryoshka representation learning** (2022), named after Russian nesting dolls, trains the model so that the *first* 256, 512, or 1024 numbers of a vector already form a good smaller embedding. You can then simply chop off the tail to save space at a modest loss of quality (Figure 13). Several current models support this, including OpenAI’s text-embedding-3, Qwen3-Embedding, and the Gemini and EmbeddingGemma models.

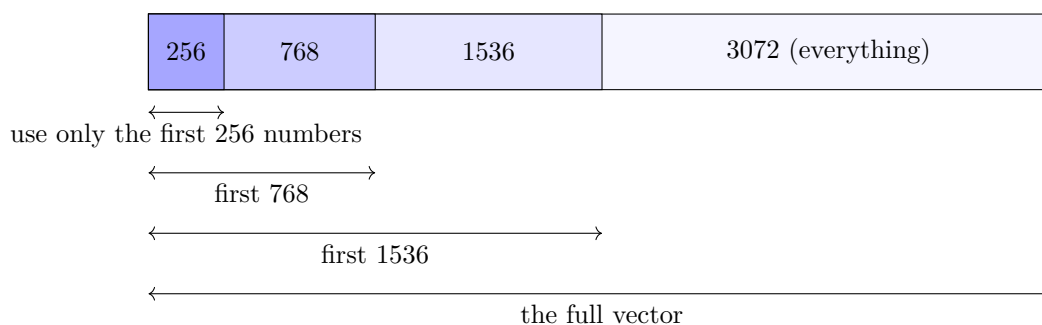
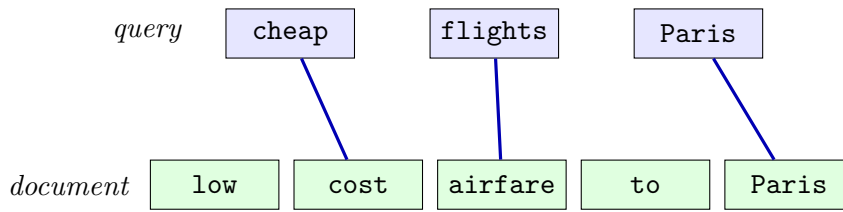


Figure 13: A Matryoshka embedding: a smaller embedding is hidden inside the larger one. (Widths are proportional to the number of dimensions.)

## 13.3 Beyond one vector: dense, sparse, and multi-vector

- **Dense** embeddings (everything so far): one vector per text. Compact and great at meaning, weaker on exact names, codes, and rare words.
- **Sparse** retrieval: classic keyword scoring (such as BM25) and learned variants. A long vector that is mostly zeros, with one entry per vocabulary word. Excellent at exact matches.
- **Multi-vector / late interaction** (ColBERT, 2020): keep one vector *per token* instead of one per text. To score a document, find for each query token its best-matching document token, and add up the scores (“MaxSim,” Figure 14). Slower and larger, but often more accurate.

- **Hybrid** systems mix several of these. For example, the open model BGE-M3 produces dense, sparse and multi-vector outputs from one network.



each query token keeps only its best match; the score is the sum of the three best-match similarities

Figure 14: Late interaction (MaxSim). Instead of squeezing the query and the document into one vector each, compare their tokens and combine the best matches.

## 13.4 Multilingual and multimodal embeddings

- **Multilingual.** Modern models place text in more than 100 languages into a single shared space, so a question in Spanish can find a document written in English.
- **Text and images together.** CLIP (2021) trained one network for images and another for captions, with the pull-together, push-apart recipe over hundreds of millions of image-caption pairs, so a photo of a dog and the sentence “a dog on a beach” land close together. This enables searching images with words.
- **Everything in one space.** In March 2026 Google released Gemini Embedding 2, described as its first *natively multimodal* embedding model: text, images, video, audio, and documents all map into a single space. Expect this to be a major direction.
- **Small and open.** In October 2026 Google followed with EmbeddingGemma 2, an open-weight (Apache 2.0) multimodal model of under one billion parameters, small enough to run on your own hardware. Its published performance claims come from the company that made it, so treat them as a starting point for your own tests.

## 13.5 The families of models you will hear about

The names change quickly, so the table lists *types* rather than a ranking. (Information gathered from public model pages and articles in October 2026; the newest release we found was EmbeddingGemma 2, announced October 6. Things move fast, so always check the current leaderboard.)

| Family                              |             | Examples                                                                          | Notes                                                                                               |
|-------------------------------------|-------------|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------|
| <b>Static word vectors</b>          |             | word2vec, GloVe, fast-Text                                                        | Tiny and fast; one vector per word; good for learning and for very small systems.                   |
| <b>Hosted models</b>                | <b>API</b>  | OpenAI text-embedding-3, Google Gemini embedding models, Cohere, Voyage           | Pay per use. Usually can shorten vectors (Matryoshka). Easy to start with.                          |
| <b>Open-weight LLM-based models</b> |             | Qwen3-Embedding (0.6B to 8B parameters, Apache 2.0 license), NV-Embed, E5-Mistral | Run on your own hardware; very strong on benchmarks; need a GPU for the larger sizes.               |
| <b>All-in-one models</b>            | <b>open</b> | BGE-M3                                                                            | Dense, sparse and multi-vector retrieval in one model; supports long inputs and over 100 languages. |
| <b>Late-interaction models</b>      |             | ColBERT family, Jina-ColBERT                                                      | Token-level matching; more accurate retrieval at higher storage cost.                               |
| <b>Multimodal models</b>            |             | CLIP, SigLIP 2, Gemini Embedding 2, EmbeddingGemma 2 (open weights)               | Put text, images and (in newer models) audio and video in one space.                                |

## 13.6 How do we compare embedding models? MTEB

Because embeddings are used for many jobs, the community built **MTEB**, the Massive Text Embedding Benchmark (2022). It scores a model on dozens of datasets covering several kinds of tasks: *retrieval* (find the relevant document), *classification*, *clustering*, *semantic textual similarity* (do two sentences mean the same?), *reranking*, and others, and it has since been extended to many more languages. A public leaderboard ranks models.

Leaderboards are useful but easy to misread. (1) The top models are often separated by differences too small to matter. (2) Rankings change month to month as new models appear. (3) Some models may be tuned to the benchmark’s own datasets. (4) Your data may be very different from the benchmark’s. **Rule of thumb:** shortlist two or three models from the leaderboard, then test them on a sample of *your own* questions and documents.

## 13.7 Practical advice

- **Never mix models.** Vectors from different models live in different “coordinate systems” and cannot be compared. If you switch models, re-embed everything.
- **Use the same model for queries and documents** (apart from optional instruction prefixes the model recommends).
- **Normalize vectors** to length 1, so that dot product equals cosine.
- **Mind the size.** One million documents with 3072 numbers each stored as 4-byte floats is about 12.3 GB; with 256 numbers (a Matryoshka cut) it is about 1 GB.

- **Mind the input limit.** Models accept only so many tokens at once (for example about 8,000 for several popular ones), so long documents must be split into chunks.
- **Consider a reranker.** A common two-stage design: fast embedding search finds the top 50 candidates, then a slower, more accurate *reranker* model reads the query and each candidate together and reorders them.

## 14 Putting embeddings to work

### 14.1 Semantic search and retrieval-augmented generation (RAG)

The most important use today. Embed every document chunk once and store the vectors in a **vector index** (a vector database). At question time, embed the question with the same model, find the nearest stored vectors, and hand those chunks to a language model so that it can answer using them. Because comparing a query against millions of vectors exactly is slow, indexes use **approximate nearest-neighbor** search (for example graph-based methods such as HNSW) that is far faster and almost as accurate.

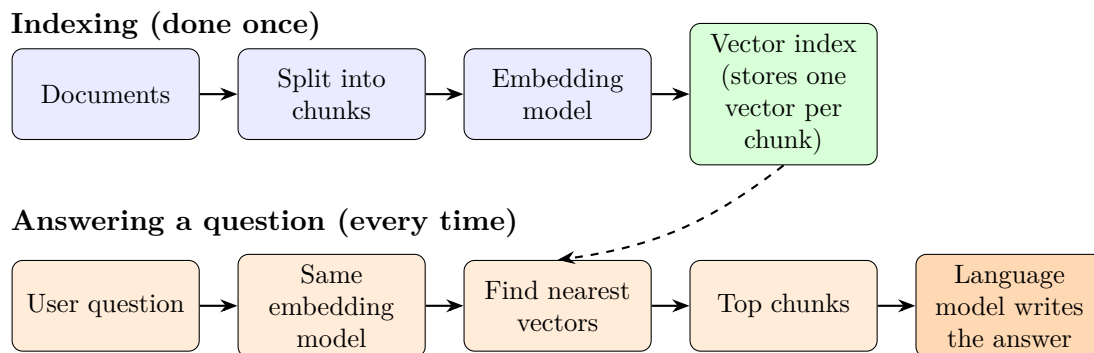


Figure 15: The retrieval-augmented generation pipeline. Both rows rely on the *same* embedding model, so that questions and chunks live in one shared space.

### 14.2 Other uses

- **Clustering and topic discovery:** group similar documents without labels.
- **Recommendation:** embed users and items, recommend the nearest items.
- **Duplicate and near-duplicate detection:** for support tickets, news, or code.
- **Classification features:** feed embeddings into a simple classifier for spam, sentiment, or intent.
- **Anomaly detection:** flag items far from everything else.
- **Visualization:** flatten embeddings to 2-D with PCA, t-SNE or UMAP, as in Figure 7, to see the structure of a collection.

Embeddings are not magic. Similar-in-meaning is not the same as “correct” or “relevant to this person’s question.” A search for “not allergic to peanuts” may retrieve documents about peanut allergies. They also carry the biases of their training data, and they can miss exact identifiers such as part numbers (one reason hybrid search exists). Always test with real examples.

## 15 The big picture

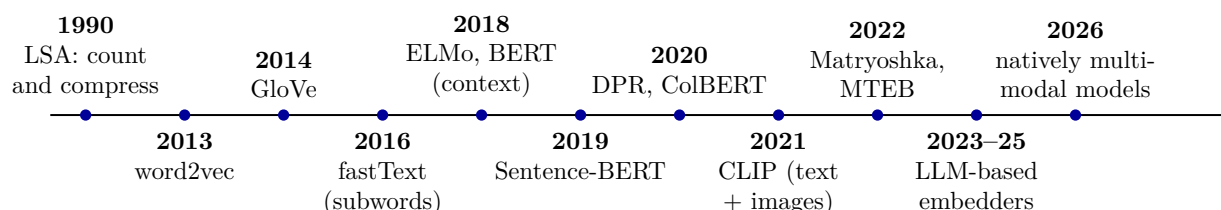


Figure 16: A (very) condensed timeline of embeddings. Spacing is by event, not by calendar year.

Look at the timeline as one story with one core idea:

- **Counting** the company of words (LSA, 1990) → **learning** the vectors directly with pull-together, push-apart (word2vec, 2013).
- Fix the one-vector-per-word problem with **context** (BERT, 2018).
- Scale the recipe from words to **whole texts** (Sentence-BERT and contrastive learning, 2019 onward), and to **images and audio**.
- Build on **giant pre-trained language models**, make vectors **flexible in size** (Matryoshka), and add **token-level matching** where accuracy matters (late interaction).

## 16 Key takeaways

1. Token ids are labels. An **embedding** replaces each by a dense vector so that meaning becomes geometry.
2. **Cosine similarity** compares the directions of two vectors, and is the standard measure of closeness.
3. Words with similar meaning appear in similar contexts. Counting neighbors (co-occurrence) already produces useful vectors.
4. **word2vec** learns short vectors by a prediction game with **negative sampling**: pull real neighbors together, push random words apart.
5. An **embedding layer** is just a table lookup (one-hot vector times a matrix). Every transformer starts with one.

6. Static vectors cannot handle multiple meanings, unknown words, or whole sentences, and they inherit biases from text.
7. **Contextual** models give each use of a word its own vector.
8. Modern text-embedding models start from large pre-trained models, train **contrastively** on huge numbers of pairs, and offer features such as instructions, shortened (Matryoshka) vectors, multiple languages, and multiple modalities.
9. The main use is **semantic search and RAG**. Benchmarks like MTEB help you shortlist models, but always test on your own data and never mix models in one index.