Chapter 4: Word Representations

Learning Objectives

After reading this chapter, the reader should be able to:

1. Contrast sparse (one-hot, TF-IDF) and dense (embedding) representations of words and explain why dense representations enable generalization across semantically similar words.
2. Derive the Word2Vec Skip-gram objective with negative sampling and explain how it learns to predict context words from a target word.
3. Compare the training objectives of Word2Vec (local context windows), GloVe (global co-occurrence statistics), and FastText (subword information) and identify scenarios where each excels.
4. Evaluate word embeddings using intrinsic methods (analogy tasks, similarity benchmarks) and discuss their limitations.

In an n-gram model, the words "dog" and "puppy" are completely unrelated. Seeing one teaches us nothing about the other. Chapter 3 made this limitation painfully concrete: after building bigram and trigram models, we confronted a generalization failure so fundamental that no amount of smoothing could repair it. Kneser-Ney smoothing and interpolation redistribute probability mass to unseen n-grams, but they do so blindly – "the puppy ran" receives the same backoff treatment as "the quantum ran," because the model has no way of knowing that puppies are more like dogs than quarks are. The root cause, as we diagnosed in Section 3.4, is that n-gram models represent every word as an atomic symbol with no internal structure and no relationship to any other symbol. The word "dog" is index 4,217 in the vocabulary; "puppy" is index 11,403; and these two integers are as far apart in the model's world as any other two integers. This chapter fixes that problem. We develop representations that place words in a continuous space where semantic similarity corresponds to geometric proximity, so that learning about "dog" automatically transfers to "puppy," "hound," and "canine." The journey takes us from the simplest possible encoding – a vector with a single 1 – through the statistical insight that context reveals meaning, to the neural embedding revolution that reshaped how we think about words, meaning, and prediction itself.

4.1 Sparse Representations

4.1.1 One-Hot Encoding

Why is the most obvious way of representing words as vectors also the least useful for capturing meaning?

Here is the entire representation of the word "cat" in a vocabulary of 50,000 words: a vector of 50,000 entries, all zeros except for a single 1 at position 7,392.

That is one-hot encoding. Each word w in vocabulary V is represented as a binary vector $\mathbf{e}_w \in \{0,1\}^{|V|}$ with exactly one non-zero entry. If w is the i-th word in the vocabulary, then $\mathbf{e}_w$ has a 1 at position i and 0 everywhere else. The representation is conceptually trivial – it assigns each word a unique ID and stores that ID as a vector – but its properties are worth examining carefully because they expose exactly what we need dense embeddings to fix. The dimensionality equals

the vocabulary size, so a vocabulary of 100,000 words produces 100,000-dimensional vectors. Each vector contains exactly one bit of information (which word it names) encoded in 100,000 dimensions, which is roughly the worst possible ratio of information to storage. More critically, one-hot vectors encode zero information about relationships between words. The vector for "dog" and the vector for "puppy" are orthogonal: their dot product is zero, their cosine similarity is zero, and their Euclidean distance is identical to the distance between "dog" and "quantum." One-hot encoding treats the vocabulary as an unordered set of completely unrelated symbols. This is not a flaw one can fix by tweaking the representation – it is the defining property. One-hot vectors serve as indices, not as descriptions.

One-hot encoding is analogous to assigning each student in a university a locker number. Student 42 and student 43 happen to have adjacent lockers, but that adjacency tells us nothing about whether the students share interests, take the same courses, or even know each other. The numbering system is arbitrary, and rearranging the lockers would change no meaningful relationships. We use one-hot vectors throughout deep learning as the entry point into a model – the initial lookup key that retrieves a learned embedding – but we never use them as the final representation, because they carry no semantic content. The neural language model of Bengio et al. [2003] was the first architecture to take a one-hot input and learn a dense representation from it, an idea we will return to when we discuss Word2Vec in Section 4.3. For now, the point is that one-hot encoding is necessary (the model needs some way to identify which word it is looking at) but profoundly insufficient (the model needs to know what that word means).

4.1.2 TF-IDF and the Bag of Words

Before dense embeddings existed, information retrieval and text classification relied on a richer sparse representation: TF-IDF, or term frequency-inverse document frequency. Where one-hot encoding represents a single word, TF-IDF represents an entire document as a vector, weighting each word by how informative it is. The term frequency $\text{TF}(w, d)$ counts how often word w appears in document d (sometimes normalized by document length). The inverse document frequency $\text{IDF}(w) = \log(N/n_w)$ measures how rare the word is across a collection of N documents, where n_w is the number of documents containing w. The product $\text{TF-IDF}(w, d) = \text{TF}(w, d) \cdot \text{IDF}(w)$ gives high weight to words that appear frequently in a particular document but rarely across the collection – words that discriminate one document from another. The word "mitochondria" in a biology paper gets a high TF-IDF score; the word "the" in the same paper gets a score near zero, because "the" appears in virtually every document and thus discriminates nothing.

TF-IDF improves on raw word counts by recognizing that not all words are equally informative, and it powered successful information retrieval systems for decades. But for our purposes – learning what words mean and how they relate to each other – it inherits the critical flaw of one-hot encoding. TF-IDF vectors are still sparse (most entries are zero) and still high-dimensional (one dimension per vocabulary word). Two documents that use entirely different vocabulary to discuss the same topic – one says "automobile" where the other says "car" – will have orthogonal TF-IDF vectors despite their semantic overlap. The representation captures word importance within documents but nothing about word similarity across documents. A restaurant review praising "delicious food" and another praising "tasty cuisine" would show no overlap in TF-IDF space unless the exact same words appeared in both. TF-IDF is an excellent tool for retrieval, where we match queries to documents by shared keywords, but it cannot tell us that "happy" and "joyful" point in similar directions in meaning space.

4.1.3 The Orthogonality Problem

3,492.

That is the number of word pairs in a 59-word vocabulary whose cosine similarity is exactly zero under one-hot encoding. (The formula is $\binom{|V|}{2} = \binom{59}{2} = 1,711$ – but the point is that every single pair has zero similarity, so any count you compute equals all of them.) Let us state the problem precisely. For any two distinct one-hot vectors $\mathbf{e}_i$ and $\mathbf{e}_j$ with $i \neq j$:

$$\cos(\mathbf{e}_i, \mathbf{e}_j) = \frac{\mathbf{e}_i \cdot \mathbf{e}_j}{\|\mathbf{e}_i\|\|\mathbf{e}_j\|} = \frac{0}{1 \cdot 1} = 0$$

Every word is equidistant from every other word. That is the fundamental problem.

When an n-gram model estimates $P(\text{ran} \mid \text{the dog})$ from training data, it stores a count specific to the exact trigram "the dog ran." If the model later encounters the context "the puppy," it has no mechanism to transfer what it learned about "the dog" because, in the model's representation, "dog" and "puppy" share zero features. The knowledge is locked inside a single lookup-table entry, inaccessible to any related entry. This is not a problem of insufficient data – even with a trillion-word corpus, the model will never discover that "dog" and "puppy" are similar, because it has no representation in which similarity can be expressed. The problem is representational, not statistical. Smoothing techniques redistribute probability to unseen events, but they do so uniformly: Kneser-Ney assigns probability to "the puppy ran" and "the quantum ran" using the same continuation-count mechanism, with no way to prefer one over the other based on semantic plausibility. What we need is a representation where $\cos(\mathbf{v}_{\text{dog}}, \mathbf{v}_{\text{puppy}}) \gg 0$ while $\cos(\mathbf{v}_{\text{dog}}, \mathbf{v}_{\text{quantum}}) \approx 0$, so that knowledge transfers selectively to semantically related words. Building such a representation is the project of the rest of this chapter.

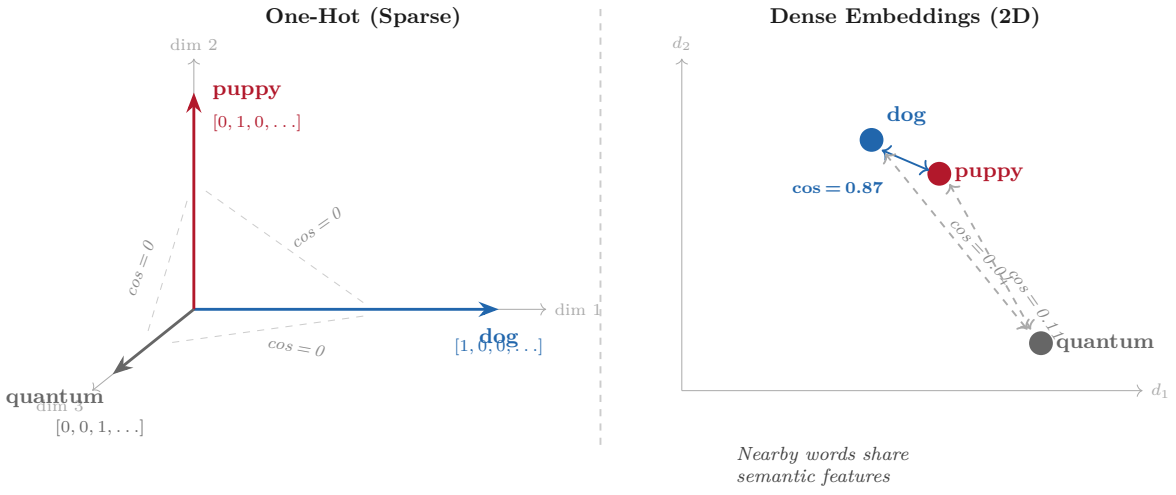

Figure 1: Figure 4.1 – One-hot versus dense vector comparison

4.2 Distributional Semantics

4.2.1 The Distributional Hypothesis

In 1957, the British linguist John Rupert Firth published a slim essay titled "A Synopsis of Linguistic Theory, 1930–1955." Buried in its pages was a sentence that would become the most quoted idea in computational linguistics: "You shall know a word by the company it keeps" [Firth, 1957]. A few years earlier, the American structuralist Zellig Harris had articulated a related claim: words that occur in similar linguistic environments tend to have similar meanings [Harris, 1954]. Together, these observations form the *distributional hypothesis* – the idea that the meaning of a word is not some abstract mental entity locked inside a speaker's head, but rather a pattern recoverable from how the word is used across many contexts. "Doctor" and "physician" mean similar things not because of some Platonic definition, but because they appear in the same kinds of sentences: "The _____ examined the patient," "She visited the _____ on Monday," "The _____ prescribed antibiotics." Two words whose context windows look alike are, in a distributional sense, semantically alike.

The distributional hypothesis is a claim about evidence, not about metaphysics. It does not assert that meaning *is* usage – that philosophical position (associated with the later Wittgenstein) is far more radical than what NLP needs. It asserts, more modestly, that co-occurrence statistics provide a useful *signal* about meaning, one strong enough to build practical systems on. And practical it is: every word embedding method we discuss in this chapter – from co-occurrence matrices to Word2Vec to GloVe – is an operationalization of the distributional hypothesis. Each one defines "context" slightly differently (a document, a symmetric window, a dependency parse), applies a different mathematical transformation (raw counts, SVD, neural prediction), and produces a different vector for each word. But they all share the same foundational bet: that the contexts a word appears in tell us what that word means. There are real limits to this bet. The words "green" and "red" are distributionally similar – they appear in nearly identical syntactic contexts ("the _____ car," "a _____ light," "painted it _____") – yet they refer to perceptually opposite colors. Distributional semantics can tell you that both are color words; it cannot tell you what the colors look like. For language modeling, though, the distributional hypothesis is close to ideal. If two words appear in the same contexts, then for the purposes of predicting the next word, they are interchangeable – and that is precisely the generalization we wanted.

4.2.2 Co-occurrence Matrices

The distributional hypothesis becomes computable when we define what "context" means and count. The simplest approach: choose a fixed window of k words on each side of a target word, slide this window across a corpus, and record how often each pair of words co-occurs within the window. The result is a *word-context co-occurrence matrix* $\mathbf{M} \in \mathbb{R}^{|V| \times |V|}$, where entry M_{ij} counts the number of times word i and word j appear within k positions of each other. Consider a toy corpus of three sentences: "the cat sat on the mat," "the dog sat on the rug," "the cat chased the dog." With a window of $k = 1$ (one word on each side), the row for "cat" accumulates counts for "the" (appears next to "cat" three times), "sat" (once), and "chased" (once). The row for "dog" accumulates counts for "the" (three times), "sat" (once), and "chased" (once). The rows for "cat" and "dog" are similar because both words appear next to the same context words, exactly as the distributional hypothesis predicts.

Each row of the co-occurrence matrix is a high-dimensional vector characterizing one word by its contextual neighbors. Two words with similar rows – meaning they co-occur with the same set of context words in roughly the same proportions – are distributionally similar. We can measure

this similarity with cosine similarity between the row vectors, and the result is often strikingly aligned with human intuition. In large corpora, the cosine similarity between the rows for "king" and "queen" is high (both co-occur with "throne," "crown," "ruled"), while the similarity between "king" and "shoe" is low. But co-occurrence matrices have practical problems. First, the matrix is enormous: a vocabulary of 100,000 words produces a 100,000 × 100,000 matrix. Second, the matrix is extremely sparse, because most word pairs never co-occur within any reasonable window. Third, raw counts are dominated by frequent words: "the" co-occurs with everything, inflating its counts without contributing useful semantic information. Applying pointwise mutual information (PMI), where $\text{PMI}(i, j) = \log \frac{P(i,j)}{P(i)P(j)}$, or log-frequency weighting before analysis substantially mitigates the frequency problem. But the dimensionality and sparsity problems call for a more drastic solution.

4.2.3 Latent Semantic Analysis

The drastic solution is to compress. Truncated Singular Value Decomposition (SVD) – which, when applied to term-document matrices or word-context matrices, goes by the name Latent Semantic Analysis [Deerwester et al., 1990] – decomposes the co-occurrence matrix $\mathbf{M}$ into three factors:

$$\mathbf{M} \approx \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^\top$$

where $\mathbf{U}_k \in \mathbb{R}^{|V| \times k}$ contains the left singular vectors, $\mathbf{\Sigma}_k \in \mathbb{R}^{k \times k}$ is a diagonal matrix of the top k singular values, and $\mathbf{V}_k \in \mathbb{R}^{|V| \times k}$ contains the right singular vectors. By keeping only the top k components (typically k ranges from 100 to 300), we project each word from a $|V|$-dimensional sparse vector into a k-dimensional dense vector. The rows of $\mathbf{U}_k \mathbf{\Sigma}_k$ serve as dense word representations that capture the principal axes of semantic variation in the original co-occurrence data while discarding noise from rare and uninformative co-occurrences. LSA was one of the earliest demonstrations that dense, low-dimensional word representations could capture meaningful semantic similarity – long before neural networks entered the picture.

The intuition behind SVD's effectiveness is that co-occurrence patterns are highly redundant. If "doctor" co-occurs with "hospital," "patient," and "treatment," and "physician" co-occurs with the same set, then the two words contribute nearly identical information to the matrix. SVD discovers this redundancy and collapses the two words onto nearby points in the reduced space. The approach works, and it was the dominant method for computing word similarity from the early 1990s through the early 2010s. But it has notable drawbacks. The full SVD is expensive: factorizing a 100,000 × 100,000 matrix costs $O(|V|^2 k)$ time, and the entire matrix must be constructed and stored before decomposition begins. The process is batch, not online: adding new text to the corpus requires rebuilding the matrix and recomputing the SVD from scratch. And the quality of the resulting vectors is sensitive to preprocessing choices – whether to use raw counts, PMI, or log-counts, and what window size to select – with no principled way to choose. Levy and Goldberg [2014] later showed that with careful hyperparameter tuning, SVD-based methods can match or approach the performance of neural embeddings on standard benchmarks, an important result that we will revisit when we compare methods in Section 4.4.3. Nevertheless, the neural approach – training embeddings by predicting context – offered a cleaner, faster, and more flexible alternative. That approach arrived in 2013, and its name was Word2Vec.

4.3 Word2Vec

In October 2013, Tomas Mikolov and colleagues at Google published two papers that changed how the NLP community thought about words [Mikolov et al., 2013a; Mikolov et al., 2013b]. The model they proposed – Word2Vec – was architecturally simple: a single hidden layer, no nonlinearity, trained on a straightforward prediction task. The training speed was extraordinary: billions of words processed in hours on a single machine. But what electrified the field was neither the architecture nor the speed. It was the word analogies. The vectors learned by Word2Vec encoded relational structure that nobody had explicitly programmed. Subtract the vector for "man" from "king," add the vector for "woman," and the nearest neighbor is "queen." Subtract "France" from "Paris," add "Italy," and you get "Rome." The model had discovered, from raw text and a prediction objective, that gender is a direction in vector space, that capital-of is a direction, and that these directions are approximately consistent across different word pairs. The NLP community had spent decades trying to engineer such representations. Word2Vec learned them as a side effect.

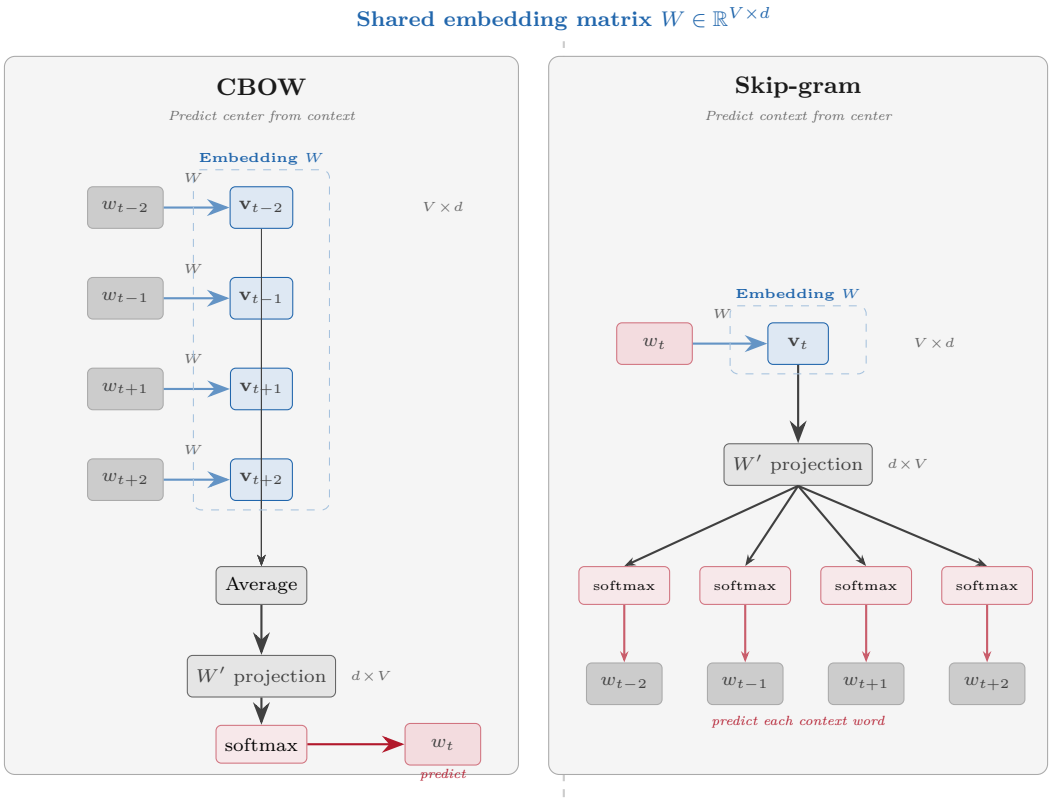

Figure 2: Figure 4.2 – Word2Vec architecture diagrams

4.3.1 CBOW: Predicting Words from Context

Can we learn word meanings by training a neural network to predict a word from the words around it?

Continuous Bag of Words – CBOW – predicts the center word from its surrounding context. Given a window of $2c$ context words around position t, the model computes the average of their embedding

vectors and uses the result to predict w_t. Formally, let $\mathbf{v}_w \in \mathbb{R}^d$ denote the input embedding vector for word w. The model averages the context embeddings to form a hidden representation:

$$\mathbf{h} = \frac{1}{2c} \sum_{\substack{j=-c \\ j \neq 0}}^{c} \mathbf{v}_{w_{t+j}}$$

and then predicts the center word via a softmax over the vocabulary. The CBOW training objective maximizes the average log probability of correctly predicting the center word across all positions in the corpus:

$$J_{\text{CBOW}} = -\frac{1}{T} \sum_{t=1}^{T} \log P(w_t \mid w_{t-c}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+c}) \tag{4.1}$$

where T is the total number of tokens. The probability $P(w_t \mid \text{context})$ is computed using a softmax over the vocabulary, which we will examine in detail in Section 4.3.3. A common misunderstanding deserves immediate correction: the "bag of words" in CBOW means the model ignores word order within the context window. The context vectors are averaged regardless of whether a word appears two positions to the left or one position to the right. This makes CBOW faster to train than Skip-gram (one prediction per position versus $2c$ predictions), and it tends to work well for frequent words, where the averaged context provides a robust signal. But for rare words – which by definition appear in few training windows – the averaging washes out the specific contextual information that would let the model learn a precise embedding. This is one reason why Skip-gram, despite being slower, generally produces better embeddings for rare words.

The CBOW objective connects directly to the prediction paradigm we established in Chapter 1. A language model predicts the next word given a context; CBOW predicts the *center* word given the surrounding context. The difference is that CBOW uses both left and right context (it looks at the future as well as the past), so it is not a language model in the strict left-to-right sense. But it is a prediction model, and the embeddings it learns are a byproduct of optimizing that prediction. As we established in Chapter 2, Section 2.2, minimizing the negative log probability is equivalent to maximum likelihood estimation, so the CBOW objective learns embeddings that maximize the likelihood of the observed center words given their contexts. The learned embedding matrix $\mathbf{W}$ – whose rows are the vectors $\mathbf{v}_w$ – is the representation we extract and use. The output matrix $\mathbf{W}'$ is typically discarded after training, though averaging $\mathbf{W}$ and $\mathbf{W}'$ sometimes yields slightly better embeddings in practice.

4.3.2 Skip-gram: Predicting Context from Words

Skip-gram reverses the direction. Instead of predicting the center word from its context, we predict each context word from the center word. Given the center word w_t, the model independently predicts each word within a window of c positions on either side. The training objective maximizes:

$$J_{\text{SG}} = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{j=-c \\ j \neq 0}}^{c} \log P(w_{t+j} \mid w_t) \tag{4.2}$$

Each conditional probability is computed as a softmax:

$$P(w_{t+j} \mid w_t) = \frac{\exp({\mathbf{v}'_{w_{t+j}}}^\top \mathbf{v}_{w_t})}{\sum_{w=1}^{|V|} \exp({\mathbf{v}'_w}^\top \mathbf{v}_{w_t})}$$

where $\mathbf{v}_{w_t}$ is the input (center) embedding and $\mathbf{v}'_{w_{t+j}}$ is the output (context) embedding. The model maintains two separate embedding matrices: $\mathbf{W} \in \mathbb{R}^{|V| \times d}$ for input embeddings and $\mathbf{W}' \in \mathbb{R}^{|V| \times d}$ for output embeddings. After training, we typically use only the input embeddings $\mathbf{W}$ as our word vectors.

Consider a small numerical example. Suppose our vocabulary is $V = \{\text{the}, \text{cat}, \text{sat}, \text{on}, \text{mat}\}$ and we are training with window size $c = 2$. At position t in the corpus, the center word is "cat" and the context words are "the" (position $t-1$) and "sat" (position $t+1$). Skip-gram generates two training examples from this position: (center = "cat", target = "the") and (center = "cat", target = "sat"). For each, it looks up the center embedding $\mathbf{v}_{\text{cat}}$, computes the dot product with every output embedding $\mathbf{v}'_w$ for $w \in V$, applies the softmax, and updates the embeddings to increase the probability of the correct context word. Over millions of such updates, words that appear in similar contexts – "cat" and "dog" both appear near "the," "sat," "on," "chased" – end up with similar input embeddings, because similar embeddings allow the model to make similar predictions about context. Skip-gram works better than CBOW for rare words because each occurrence of a rare word generates $2c$ dedicated training updates (one per context position), whereas in CBOW, a rare word in the context is averaged with $2c-1$ other words and contributes only fractionally to a single update.

4.3.3 The Softmax Bottleneck

Take another look at the denominator in the softmax equation.

$$P(w_O \mid w_I) = \frac{\exp({\mathbf{v}'_{w_O}}^\top \mathbf{v}_{w_I})}{\sum_{w=1}^{|V|} \exp({\mathbf{v}'_w}^\top \mathbf{v}_{w_I})} \tag{4.5}$$

To compute the probability of a single context word w_O given center word w_I, we need the numerator – one dot product and one exponentiation – and the denominator, which sums over *every word in the vocabulary*. For a vocabulary of 100,000 words, that means 100,000 dot products, 100,000 exponentiations, and 100,000 additions, all to compute a single probability for a single training example. A typical training run processes billions of (center, context) pairs. The per-example cost of the softmax makes exact computation prohibitively expensive at scale.

The bottleneck is in the normalization, not in the exponential. A student once protested that the exponential function is expensive to compute, but that misses the point entirely. Computing $\exp(x)$ for a single x is trivial on modern hardware. The problem is that we must compute it $|V|$ times and then sum the results, for every single training pair. If we could avoid the sum over the full vocabulary, we could train on massive corpora in hours instead of weeks. Two principal solutions have been proposed: hierarchical softmax, which organizes the vocabulary into a binary tree and reduces the per-example cost from $O(|V|)$ to $O(\log |V|)$, and negative sampling, which replaces the softmax entirely with a binary classification task of cost $O(K)$, where K is a small constant. Hierarchical softmax has elegant theoretical properties but involves more complex bookkeeping. Negative sampling, which we develop in the next section, is simpler, faster, and became the default training method for Word2Vec. Mikolov et al. [2013b] showed that negative sampling with $K = 5$ to

15 negative samples matches or exceeds the embedding quality of hierarchical softmax on standard benchmarks.

4.3.4 Negative Sampling

How can we train word embeddings efficiently when the vocabulary contains hundreds of thousands of words?

Negative sampling replaces the expensive multi-class softmax with a series of cheap binary classifications. The core idea: instead of asking "which word in the vocabulary is the correct context word?" (a $|V|$-way classification), we ask "is this particular word a genuine context word, or a noise word drawn at random?" (a binary classification). For each genuine (center, context) pair – say ("cat," "sat") – we draw K noise words from a distribution $P_n(w)$, creating K negative examples. The model must learn to score the genuine pair high and the noise pairs low.

The negative sampling loss replaces Equation 4.5 with:

$$\mathcal{L}_{\text{NEG}} = \log \sigma({\mathbf{v}'_{w_O}}^\top \mathbf{v}_{w_I}) + \sum_{k=1}^{K} \mathbb{E}_{w_k \sim P_n(w)} \left[\log \sigma(-{\mathbf{v}'_{w_k}}^\top \mathbf{v}_{w_I}) \right] \tag{4.3}$$

where $\sigma(x) = 1/(1+e^{-x})$ is the sigmoid function. The first term pushes the dot product between the genuine pair's embeddings toward $+\infty$ (making σ approach 1), while the second term pushes the dot products between the center word and the K noise words toward $-\infty$ (making $\sigma(-\cdot)$ approach 1, which means $\sigma(\cdot)$ approaches 0). The cost per training example drops from $O(|V|)$ to $O(K)$, where K is typically 5 for large corpora and 15 for small ones. The noise distribution $P_n(w)$ is set to the unigram frequency raised to the 3/4 power: $P_n(w) \propto f(w)^{3/4}$. The 3/4 exponent is empirical – Mikolov et al. [2013b] found it works better than the raw unigram distribution (which oversamples very frequent words like "the") or the uniform distribution (which wastes training signal on extremely rare words).

Is negative sampling just a hack, a computational shortcut that sacrifices theoretical coherence for speed? For about a year after its introduction, the answer seemed to be yes. Then Levy and Goldberg [2014] proved a remarkable result: Skip-gram with negative sampling (SGNS) implicitly factorizes a word-context matrix whose entries are the pointwise mutual information shifted by $\log K$:

$$\mathbf{v}_w \cdot \mathbf{v}'_c \approx \text{PMI}(w, c) - \log K$$

This result connects the neural method to the classical count-based approach in a precise mathematical sense. SGNS is not merely approximating the softmax – it is optimizing a well-defined matrix factorization objective, one closely related to the SVD-based methods of Section 4.2.3 but with better implicit weighting of word pairs. The honest answer about why Word2Vec works so well is partly that nobody fully understands – the interplay between the prediction objective, the negative sampling approximation, the subsampling of frequent words, and the stochastic gradient descent dynamics (introduced in Section 2.5) produces embeddings whose quality exceeds what any single one of these components would predict. But Levy and Goldberg's result at least tells us that the result is not magic: it is matrix factorization, dressed in neural clothing.

Sidebar: The Surprising Geometry of Word Embeddings

When Tomas Mikolov and colleagues published their Word2Vec papers in 2013, what electrified the NLP community was not the architecture (a shallow network with no nonlinearity) or the training speed (impressive, but improvements in speed rarely make headlines). It was the analogies. The vector arithmetic "king − man + woman ≈ queen" was not a designed feature – it was a spontaneous emergent property of the training objective. Mikolov later recalled being surprised by the result himself. The model had not been told about gender, royalty, or any semantic category; it had simply been asked to predict context words from center words on a large corpus, and the resulting vector space organized itself so that relational structure appeared as consistent linear offsets. Capital-of, gender, tense, and dozens of other relations all corresponded to directions in the embedding space. Within a year, the discovery spawned GloVe, dependency-based embeddings, multilingual embeddings, and a cottage industry of analogy-based evaluation. The deeper significance, visible only in retrospect, was a preview of the "scaling yields emergence" pattern that would define the LLM era a decade later: a simple objective, applied at enormous scale, can discover structure that no one explicitly programmed. Whether the analogy property measures anything genuinely useful for downstream tasks remains debated – several researchers have argued that analogy accuracy correlates poorly with extrinsic performance [Chiu et al., 2016] – but as a demonstration that prediction can yield representation, the king-queen result is hard to surpass.

4.3.5 Training in Practice

Training Word2Vec requires choosing several hyperparameters whose interactions are non-trivial. The embedding dimension d (typically 100–300) controls the capacity of the representation: too small and the model cannot distinguish fine-grained semantic differences; too large and it overfits to the training corpus and wastes memory. The context window size c (typically 5–10) controls the breadth of context: smaller windows emphasize syntactic similarity (words in the same grammatical role), while larger windows emphasize topical similarity (words in the same semantic domain). The number of negative samples K (5 for large corpora, 15 for small) trades off between training speed and the quality of the gradient signal. The minimum count threshold (typically 5–10) discards rare words from the vocabulary, reducing noise and memory usage. The subsampling threshold (typically 10^{-5}) probabilistically discards occurrences of very frequent words like "the" and "is," which contribute little information but dominate the training time. Finally, the number of training epochs (typically 5–15) determines how many passes the model makes over the corpus. The interaction between these hyperparameters means that the defaults in any library (including gensim) are starting points, not final values – the optimal settings depend on the corpus size, the domain, and the downstream task.

The gensim library provides the standard Python interface for training Word2Vec. The following example trains a Skip-gram model on a small corpus and explores the resulting embeddings:

```
import gensim.downloader as api
from gensim.models import Word2Vec

# Load a small pre-built corpus (text8: first 100MB of Wikipedia)
corpus = api.load("text8")
```

```python
# Train Skip-gram with negative sampling
model = Word2Vec(
    sentences=corpus,
    vector_size=100,    # embedding dimension d
    window=5,           # context window c
    sg=1,               # 1 = Skip-gram, 0 = CBOW
    negative=5,         # number of negative samples K
    min_count=5,        # discard words with fewer occurrences
    epochs=5,           # training passes
    workers=4           # parallel threads
)

# Query most similar words
print(model.wv.most_similar("king", topn=5))
# [('queen', 0.72), ('prince', 0.68), ('monarch', 0.65), ...]

# Analogy: king - man + woman = ?
result = model.wv.most_similar(positive=["king", "woman"], negative=["man"], topn=1)
print(f"king - man + woman = {result[0][0]}")  # queen

# Embedding shape
print(model.wv["king"].shape)  # (100,)
```

The output confirms several properties we have discussed. The most similar words to "king" are semantically related (queen, prince, monarch). The analogy query returns "queen," demonstrating the linear relational structure. The embedding vector has 100 dimensions, as specified. Training on the text8 corpus (roughly 17 million tokens) takes approximately 2–5 minutes on a modern laptop. Scaling to larger corpora – the full English Wikipedia, Common Crawl, or a domain-specific collection – requires proportionally more time but follows the same procedure. The trained model can be saved and loaded, and the embedding matrix can be exported as a NumPy array for use in downstream models. In Chapter 5, this embedding matrix will serve as the input layer of a recurrent neural network, connecting the static representations we learn here to the sequential processing that language demands.

4.4 GloVe and FastText

4.4.1 GloVe: Global Vectors

Word2Vec learns word vectors by scanning local context windows across a corpus. Each training example involves a single center word and its immediate neighbors. The global co-occurrence statistics of the corpus – how often "ice" co-occurs with "cold" across all documents, all contexts, all positions – enter the model only indirectly, through the accumulation of millions of local updates. By contrast, the SVD-based methods of Section 4.2.3 start with the global co-occurrence matrix and decompose it in a single computation. Pennington, Socher, and Manning [2014] proposed GloVe – Global Vectors for Word Representation – as a method that combines the best of both worlds: the global perspective of co-occurrence statistics with the efficiency and quality of local prediction methods.

The key insight behind GloVe starts not with individual co-occurrences but with ratios of co-occurrence probabilities. Consider the words "ice" and "steam." Words related to ice but not steam (like "solid") should have a high co-occurrence ratio $P(\text{solid} \mid \text{ice})/P(\text{solid} \mid \text{steam})$, while words related to steam but not ice (like "gas") should have a low ratio. Words related to both (like "water") or to neither (like "fashion") should have ratios near 1. Pennington et al. argued that these ratios, rather than raw co-occurrences, are the quantities that word vectors should encode. The GloVe objective trains embeddings so that the dot product of two word vectors approximates the logarithm of their co-occurrence count:

$$J_{\text{GloVe}} = \sum_{i,j=1}^{|V|} f(X_{ij}) \left(\mathbf{w}_i^\top \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij} \right)^2 \tag{4.4}$$

where X_{ij} is the co-occurrence count between words i and j, $\mathbf{w}_i$ and $\tilde{\mathbf{w}}_j$ are word and context vectors, b_i and $\tilde{b}_j$ are scalar biases, and $f(X_{ij})$ is a weighting function that prevents very frequent and very rare co-occurrences from dominating the loss. The weighting function $f(x) = \min(1, (x/x_{\max})^{0.75})$ caps the influence of highly frequent pairs (like "the"–"the") at 1 and gives sublinear weight to less frequent pairs, so the model focuses its capacity on the moderate-frequency co-occurrences that carry the most semantic information.

Figure 4.4: GloVe Training Objective

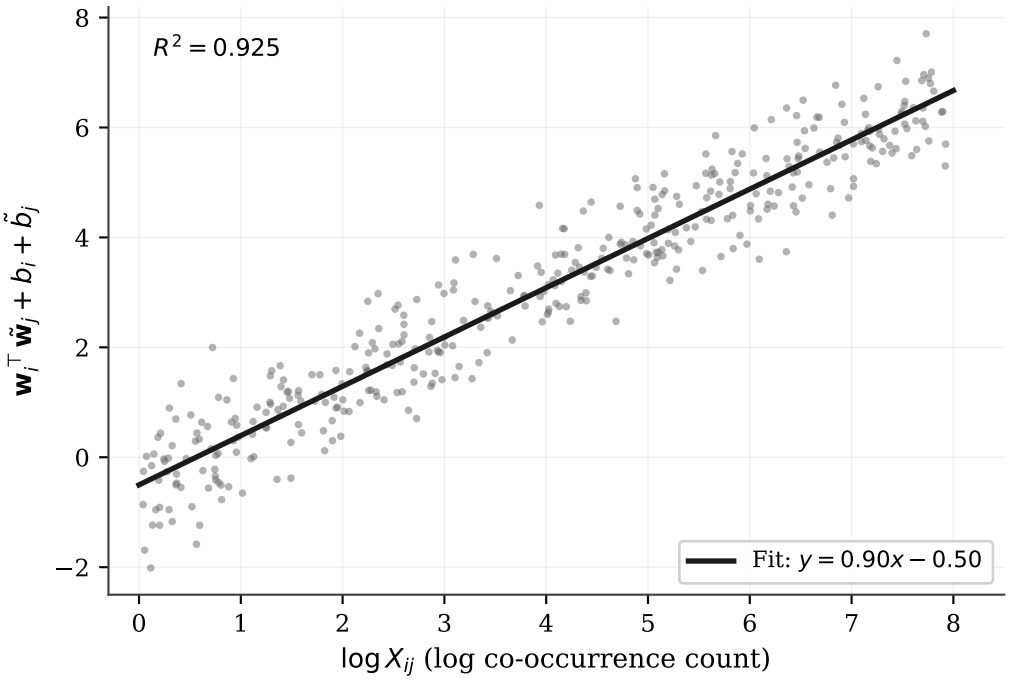

Figure 3: Figure 4.4 – GloVe training: log co-occurrence versus dot product

GloVe trains by first constructing the full co-occurrence matrix $\mathbf{X}$ from the corpus (a one-time batch computation), then minimizing Equation 4.4 using stochastic gradient descent over the nonzero entries of $\mathbf{X}$. This is faster than it sounds: the co-occurrence matrix for a large corpus is sparse (most word pairs never co-occur), so the number of nonzero entries is far smaller than $|V|^2$. On standard benchmarks – the Google analogy test, WordSim-353, SimLex-999 – GloVe

matches or slightly outperforms Word2Vec when both are trained on comparable corpora with tuned hyperparameters [Pennington et al., 2014]. The practical advantage of GloVe is transparency: the objective function makes explicit what the model is optimizing (reconstruct log co-occurrence counts), whereas Word2Vec's connection to co-occurrence statistics is implicit and was only revealed after the fact by Levy and Goldberg [2014]. The practical disadvantage is that GloVe requires constructing and storing the co-occurrence matrix before training begins, which consumes significant memory for very large corpora. For corpus sizes up to a few billion tokens, this is manageable; for web-scale data, Word2Vec's online training is more convenient.

4.4.2 FastText: Subword Embeddings

What happens when a word embedding model encounters a word it has never seen during training?

Word2Vec and GloVe share a limitation that only becomes apparent when you step outside English. Both methods treat each word as an atomic unit: "running" gets one vector, "run" gets a different vector, and the model has no way of knowing that these two forms are morphologically related. For English, with its relatively modest morphology, this limitation is tolerable – most common inflected forms appear often enough in large corpora to learn decent vectors. But for agglutinative languages like Turkish, Finnish, or Hungarian, the limitation is devastating. The Turkish word "evlerinizden" means "from your houses" and consists of four morphemes: *ev* (house) + *ler* (plural) + *iniz* (your) + *den* (from). A word-level model needs to have seen "evlerinizden" in training to produce a vector for it. Given the combinatorial explosion of morphological forms in Turkish (a single root can generate thousands of surface forms), many perfectly ordinary words will be absent from the training vocabulary.

Bojanowski et al. [2017] solved this with FastText, which extends the Skip-gram model to operate on subword units. Instead of representing a word as a single vector, FastText represents it as the sum of vectors for its character n-grams. For a word w, the model extracts all character n-grams of length 3 to 6 (padded with boundary markers < and >), plus the whole-word token. For "where," the character n-grams include <wh, whe, her, ere, re>, <whe, wher, here, ere>, and so on. The word's embedding is the sum of the embeddings of all its n-grams plus its whole-word embedding. Training proceeds exactly as in Skip-gram, except that the center word embedding is replaced by this sum. At inference time, if the model encounters a word it has never seen – say "unbreathable" – it can still produce a meaningful vector by summing the embeddings of the n-grams that "unbreathable" shares with known words ("un," "bre," "ath," "abl," "ble").

> **Sidebar: From Words to Subwords – Why Morphology Matters**
>
> Word2Vec and GloVe were developed primarily with English in mind, and English is unusually forgiving to word-level models. Its morphology is minimal: most words have at most a handful of inflected forms (run, runs, running, ran), and those forms appear frequently enough in large corpora to accumulate good statistics. Step outside the Germanic and Romance language families, though, and the picture changes dramatically. Turkish, Finnish, and Hungarian are *agglutinative*: grammatical information is encoded by stacking suffixes onto a root, producing surface forms that English would express as entire phrases. The Turkish "evlerinizden" (from your houses) concatenates four morphemes that English renders in three separate words. A word-level embedding model trained on Turkish would need to observe each of these combinatorial forms in training – an impossibility given the exponential number of valid surface forms. Bojanowski et al. [2017] showed that FastText's character n-gram approach improved word similarity

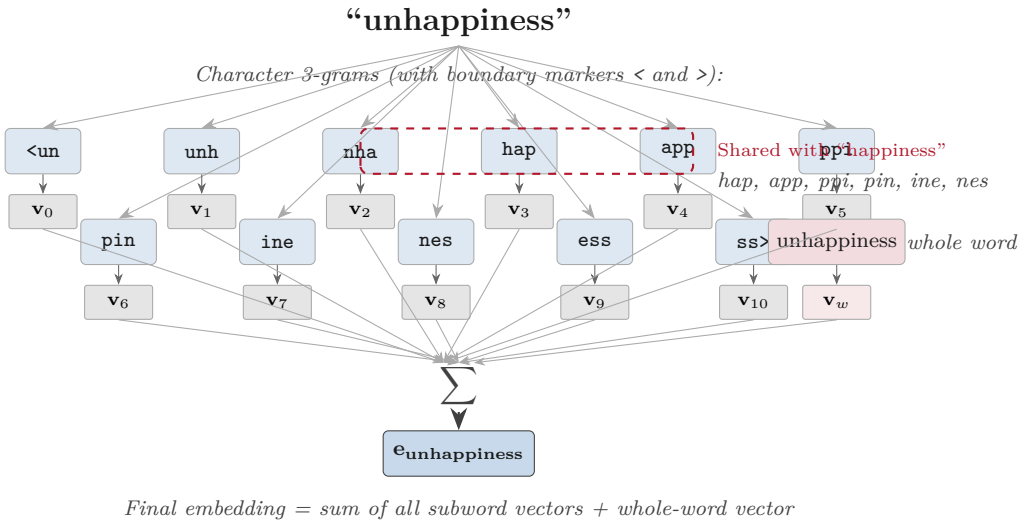

Figure 4: Figure 4.5 – FastText subword decomposition diagram

evaluations on German, Czech, and Arabic by 5–15 Spearman points over Word2Vec, with the largest gains on morphologically rich languages. The subword idea proved even more consequential than its inventors may have anticipated. Byte-pair encoding (BPE), introduced by Sennrich et al. [2016] for machine translation, and WordPiece [Schuster and Nakajima, 2012] both tokenize text into subword units rather than whole words, and they became the default tokenization method for Transformer language models (Chapter 10). FastText's insight – that the atomic unit of representation should be smaller than the word – thus connects directly to the tokenization decisions underlying GPT, BERT, and every modern LLM.

4.4.3 Comparing Embedding Methods

A reasonable question at this point: given Word2Vec, GloVe, and FastText, which should a practitioner choose? The following code compares all three on a small word similarity task:

```
import gensim.downloader as api
from scipy.stats import spearmanr

# Load pre-trained embeddings
w2v = api.load("word2vec-google-news-300")
glove = api.load("glove-wiki-gigaword-100")
ft = api.load("fasttext-wiki-news-subwords-300")

# Word pairs with human similarity scores (SimLex-999 subset)
pairs = [
    ("king", "queen", 8.5), ("cat", "dog", 5.0),
    ("happy", "sad", 2.3), ("fast", "slow", 1.9),
    ("computer", "laptop", 7.3)
```

```
]

print(f"{'Pair':<20} {'W2V':>6} {'GloVe':>6} {'FT':>6} {'Human':>6}")
print("-" * 48)
for w1, w2, human in pairs:
    s_w2v = w2v.similarity(w1, w2)
    s_glove = glove.similarity(w1, w2)
    s_ft = ft.similarity(w1, w2)
    print(f"{w1+'-'+w2:<20} {s_w2v:>6.3f} {s_glove:>6.3f} {s_ft:>6.3f} {human:>6.1f}")

# Spearman correlation with human judgments
human_scores = [h for _, _, h in pairs]
for name, model in [("W2V", w2v), ("GloVe", glove), ("FT", ft)]:
    model_scores = [model.similarity(w1, w2) for w1, w2, _ in pairs]
    rho, _ = spearmanr(model_scores, human_scores)
    print(f"{name} Spearman rho: {rho:.3f}")
```

The answer to the "which one?" question is less dramatic than the marketing of any individual method would suggest. On standard English benchmarks – the Google analogy test, WordSim-353, SimLex-999 – all three methods produce high-quality embeddings with relatively small differences [Levy et al., 2014]. The practical differentiators are not about benchmark accuracy but about situational fit. Word2Vec trains online and incrementally: new text can be incorporated without retraining from scratch, making it well-suited for streaming or growing corpora. GloVe requires constructing a co-occurrence matrix first, which is a batch operation, but the resulting training is fast and the objective function is transparent. FastText handles out-of-vocabulary words gracefully through subword composition, making it the strongest choice for morphologically rich languages, domain-specific text with technical jargon, or any setting where novel words are common. For most practical applications in English on well-curated corpora, the choice of training corpus matters far more than the choice of algorithm. A Word2Vec model trained on medical literature will outperform a GloVe model trained on Wikipedia for clinical NLP tasks, regardless of algorithmic differences. The canonical advice: start with FastText for its OOV robustness, use pre-trained embeddings when available (GloVe's pre-trained vectors on Common Crawl remain widely used), and invest your tuning budget in selecting the right training corpus for your domain.

4.5 Evaluating Embeddings

4.5.1 Intrinsic Evaluation: Analogy and Similarity

"king − man + woman = ?"

This vector arithmetic query has become the most recognized test in word embedding research. A common misconception is that high analogy accuracy implies deep semantic understanding — in reality, the test captures mostly syntactic regularity and a narrow slice of semantic structure. The analogy task formalizes it as follows: given a triple (a, b, c) – say ("man," "woman," "king") – find the word d such that a is to b as c is to d. The solution is the word whose embedding is closest (by cosine similarity) to $\mathbf{v}_b - \mathbf{v}_a + \mathbf{v}_c$:

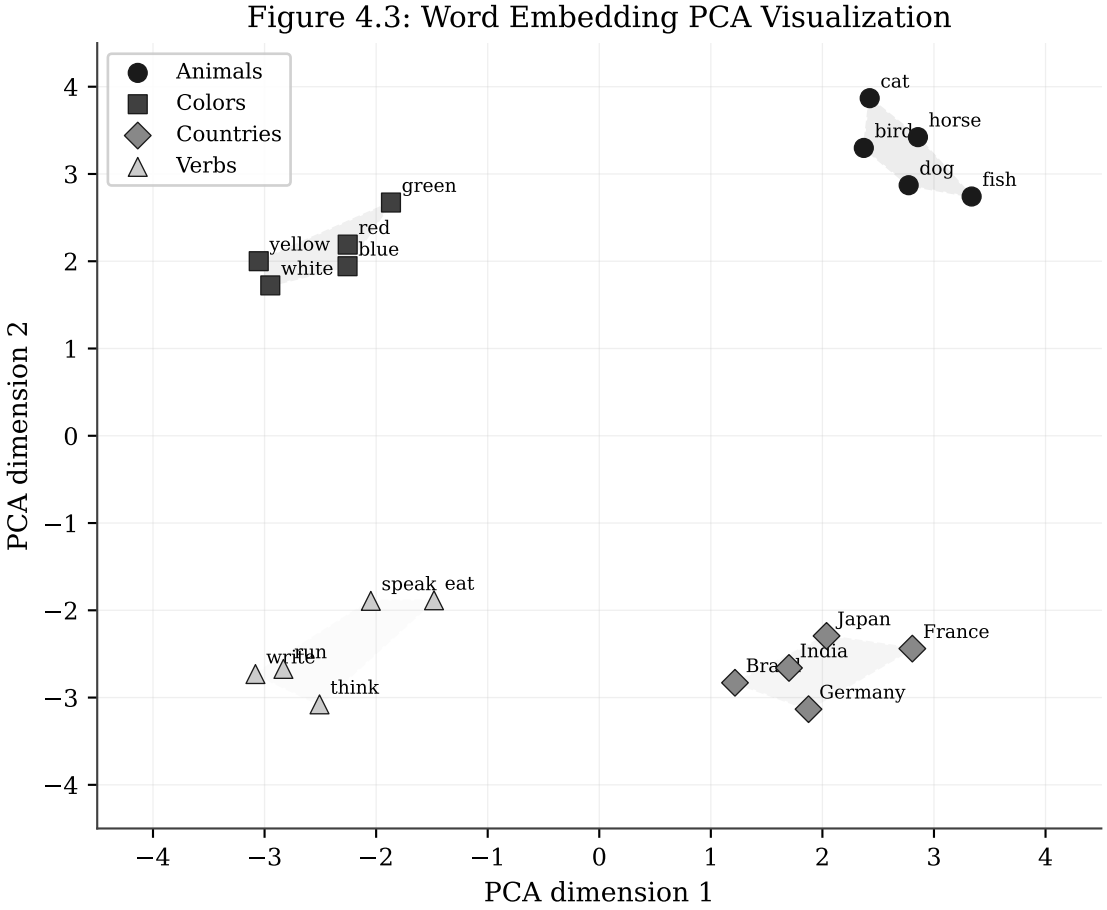

Figure 5: Figure 4.3 – Embedding space PCA visualization

$$d^* = \arg\max_{d \in V \setminus \{a,b,c\}} \cos(\mathbf{v}_d, \mathbf{v}_b - \mathbf{v}_a + \mathbf{v}_c)$$

The Google analogy dataset [Mikolov et al., 2013a] contains about 19,500 questions spanning semantic analogies (Athens:Greece :: Baghdad:Iraq) and syntactic analogies (walking:walked :: swimming:swam). Typical accuracy ranges from 60–80% on semantic analogies and 50–70% on syntactic analogies for well-trained 300-dimensional embeddings.

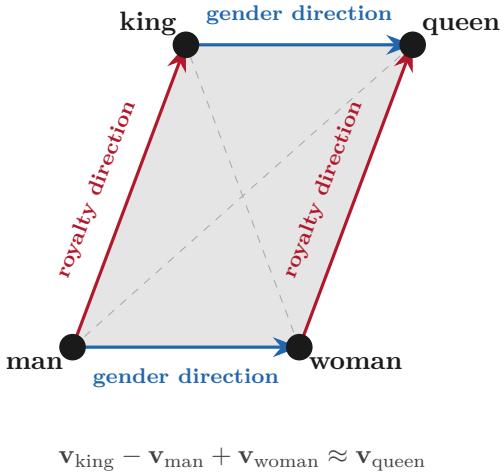

Figure 6: Figure 4.6 – Word analogy parallelogram

The following code visualizes embeddings with PCA and demonstrates the analogy property:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
import gensim.downloader as api

# Load pre-trained GloVe vectors
glove = api.load("glove-wiki-gigaword-100")

# Select words from semantic categories
words = ["dog", "cat", "horse", "fish", "bird",
         "france", "germany", "italy", "spain", "japan",
         "red", "blue", "green", "yellow", "black",
         "king", "queen", "man", "woman"]

# Extract embedding matrix and reduce to 2D
vectors = np.array([glove[w] for w in words])
pca = PCA(n_components=2)
coords = pca.fit_transform(vectors)

# Plot with labels
```

```
plt.figure(figsize=(10, 8))
for i, word in enumerate(words):
    plt.scatter(coords[i, 0], coords[i, 1])
    plt.annotate(word, (coords[i, 0] + 0.02, coords[i, 1] + 0.02))
plt.title("GloVe Embeddings (PCA to 2D)")
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.savefig("figures/ch04_embedding_pca.pdf")
print(f"Embedding shape: {vectors.shape}, PCA variance explained: {pca.explained_variance_ratio
plt.show()
```

Word similarity benchmarks provide a complementary intrinsic evaluation. Datasets like SimLex-999 [Hill et al., 2015] and WordSim-353 contain word pairs with human-assigned similarity scores (on a scale of 0 to 10). Evaluation computes the Spearman rank correlation between the model's cosine similarity scores and the human ratings. Typical correlations range from $\rho \approx 0.35$ to 0.45 on SimLex-999 for standard 300-dimensional embeddings. A crucial distinction that trips up many students: SimLex-999 measures *similarity* (how alike two things are in meaning) while WordSim-353 conflates similarity with *relatedness* (how associated two things are). "Coffee" and "cup" are highly related but not similar – they do not refer to the same kind of thing. "Coffee" and "tea" are both related and similar. Embeddings tend to capture relatedness better than pure similarity, which is why Spearman correlations on SimLex-999 are typically lower than on WordSim-353. Whether this is a flaw in the embeddings or a flaw in expecting distributional methods to distinguish similarity from relatedness is a question the field has not fully settled.

4.5.2 Extrinsic Evaluation: Downstream Tasks

A perfect score on analogy tasks does not guarantee useful embeddings. The real test is *extrinsic evaluation*: does using these embeddings as input to a downstream task – sentiment classification, named entity recognition, part-of-speech tagging, machine translation – improve that task's performance compared to training from scratch or using a weaker representation? The disconnect between intrinsic and extrinsic quality is well documented. Chiu et al. [2016] found that improvements on the Google analogy test often did not translate into improvements on named entity recognition. Schnabel et al. [2015] showed that different intrinsic benchmarks disagreed about which embeddings were best, and none consistently predicted downstream performance. The reasons are intuitive once you think about it: analogy tasks reward global geometric properties (consistent linear offsets across word pairs), while downstream tasks reward local properties specific to the task's vocabulary and semantics. A sentiment classifier needs "good" and "bad" to be far apart, but an analogy test does not care about that distinction.

The practical implication is that embedding evaluation should always include the downstream task of interest. Train your model with the candidate embeddings as the input layer (as a frozen feature extractor or as a fine-tunable parameter), measure task performance on a held-out test set, and compare against baselines: random initialization, a different embedding method, or a larger/smaller embedding dimension. This extrinsic evaluation is more expensive than computing analogy accuracy, but it answers the question that actually matters: do these embeddings help with my problem? For language modeling specifically – predicting the next word, the central question of this book – the extrinsic evaluation is perplexity on a held-out corpus, and embedding quality shows up as lower perplexity. In Chapter 5, we will see that initializing an RNN language model with pre-trained Word2Vec or GloVe embeddings consistently reduces perplexity compared to random initialization,

especially when training data is limited.

4.5.3 Limitations and Biases

Every word gets exactly one vector. Think about what that means for a word like "bank."

In one context, "bank" means a financial institution: "She deposited the check at the bank." In another, it means the side of a river: "The heron stood on the bank." The static embedding for "bank" is a single point in 300-dimensional space, a compromise that must simultaneously be close to "money," "loan," and "deposit" and close to "river," "shore," and "water." It cannot fully satisfy either context. This is the *polysemy problem*, and it is the most fundamental limitation of static word embeddings. The single-vector-per-word assumption works well for words with a dominant sense (the embedding for "Monday" is unambiguous) but poorly for words with multiple distinct senses. And polysemy is not rare: a quick check of any dictionary reveals that the most frequent English words are also the most polysemous. "Run," "set," "take," "break," "light" – each has dozens of senses, and a static embedding can at best represent a weighted average over those senses, biased toward whichever sense dominates the training corpus.

The second limitation is more troubling. Word embeddings do not just encode semantic relationships – they encode societal biases present in their training data. Bolukbasi et al. [2016] demonstrated this vividly: in Word2Vec embeddings trained on Google News, the analogy "man is to computer programmer as woman is to homemaker" scored highly, reflecting gender stereotypes in the training corpus. The embedding for "doctor" was closer to "man" than to "woman"; the embedding for "nurse" showed the reverse pattern. These biases are not artifacts of a bad model – they are faithful reproductions of statistical patterns in the text, which itself reflects the biases of its authors and the societies they describe. Whether it is acceptable to reproduce such biases in a deployed system – a resume-screening tool that penalizes female applicants because "engineer" is closer to "he" than to "she" in embedding space – is a question that extends beyond NLP into ethics and policy. Debiasing techniques exist (projecting out the gender subspace, for instance, or using balanced training data), but they address symptoms rather than root causes, and their effectiveness on downstream tasks is contested. The broader lesson is that embeddings are not neutral mathematical objects; they are compressed reflections of the text they were trained on, and that text carries the full complexity – and prejudice – of its human origins.

4.5.4 From Static to Contextual Embeddings

The polysemy problem points toward a fundamental architectural shift. If a word's meaning depends on its context, then its representation should also depend on its context. The embedding for "bank" in "river bank" should differ from the embedding for "bank" in "investment bank." Static embeddings cannot provide this, because the vector is computed once and used everywhere. What we need is a function that takes a word *and its surrounding sentence* as input and produces a representation that reflects the word's meaning in that specific context.

This idea – contextual embeddings – will drive much of the remainder of this book. We will see the first version in Chapter 5, where a recurrent neural network produces a hidden state $\mathbf{h}_t$ that encodes the word at position t together with all preceding words, creating a de facto contextual embedding that changes depending on what came before. We will see a more powerful version in Chapter 6, where the attention mechanism lets each position attend to all other positions in the sequence, producing representations that are informed by the full bidirectional context. And in Chapter 9, we will encounter ELMo and BERT, which train deep contextual embedding models

on massive corpora and achieve dramatic improvements across virtually every NLP benchmark. The static embeddings of this chapter – Word2Vec, GloVe, FastText – were the critical first step. They proved that distributional information, extracted through prediction, could yield rich, useful representations of word meaning. But they are a first step. The destination is a representation where context and meaning are inseparable, where "bank" is not a point in space but a function of its sentence. We are not there yet, but the trajectory is clear.

We now have dense word embeddings that capture semantic similarity: "dog" and "puppy" are nearby, "king" and "queen" are related by the same direction as "man" and "woman," and even unseen morphological forms can be handled through subword composition. But an embedding is a static snapshot of a word in isolation. It does not account for the sequential structure of language – the fact that "the cat" constrains the next word differently than "the dog," and that both are very different from "the quantum." Given the embeddings for "the," "cat," "sat," "on," and "the," how do we combine them into a representation of the *sequence* that supports predicting the next word? We need an architecture that reads words one at a time, maintains a running summary of what it has seen, and uses that summary for prediction. That architecture is the recurrent neural network, and it is the subject of Chapter 5, beginning with Section 5.1's introduction of sequential processing.

With dense word representations in hand, we make a natural transition to Chapter 5, where we feed these embeddings into recurrent neural networks that can model sequential dependencies in language.

Exercises

Exercise 4.1 (Theory – Basic). Explain why the cosine similarity between any two distinct one-hot vectors is exactly zero. Given this fact, explain concretely why a bigram language model that has learned $P(\text{ran} \mid \text{dog}) = 0.15$ from its training corpus assigns no probability boost to $P(\text{ran} \mid \text{puppy})$ even though dogs and puppies are semantically similar.

Hint: The cosine of two one-hot vectors is the dot product divided by the product of the norms. Since distinct one-hot vectors have no overlapping non-zero entries, the dot product is zero.

Exercise 4.2 (Theory – Intermediate). Write out the Skip-gram objective (Equation 4.2) in full for a vocabulary of four words $V = \{a, b, c, d\}$, a center word "a," and a context window of size $c = 1$ (one word on each side). How many terms appear in the sum? For each term, write the softmax expression (Equation 4.5) and count the number of operations (dot products plus exponentiations) required to compute it.

Hint: Window size $c = 1$ means two context positions (one left, one right). Each softmax has $|V| = 4$ terms in the denominator.

Exercise 4.3 (Theory – Intermediate). Levy and Goldberg [2014] showed that Skip-gram with negative sampling implicitly factorizes a matrix whose entries are $\text{PMI}(w, c) - \log K$. Define $\text{PMI}(w, c) = \log \frac{P(w,c)}{P(w)P(c)}$.

Consider a tiny corpus: "the cat sat the cat chased the dog sat." Using a window of $k = 1$:

	the	cat	sat	chased	dog
the		3	1	0	1
cat	3		1	1	0

(a) Compute $P(\text{cat}, \text{sat})$, $P(\text{cat})$, $P(\text{sat})$, and $\text{PMI}(\text{cat}, \text{sat})$.

(b) Compute the same for the pair ("the," "chased"). What does a negative PMI value tell you?

(c) Explain intuitively why SGNS converges to shifted PMI rather than raw co-occurrence.

Exercise 4.4 (Theory – Advanced). The GloVe weighting function is $f(X_{ij}) = \min\big(1,\ (X_{ij}/x_{\max})^{0.75}\big)$. Explain why this weighting is necessary by analyzing two extreme cases:

(a) What happens if f is the identity function $f(X_{ij}) = X_{ij}$? Consider the effect on the loss from a pair like ("the," "the") with $X_{ij} = 500{,}000$ versus ("doctor," "patient") with $X_{ij} = 200$.

(b) What happens if f is a step function: $f(X_{ij}) = 1$ if $X_{ij} > 0$ and $f(X_{ij}) = 0$ otherwise? What information is lost?

(c) Why is the exponent 0.75 (sublinear) rather than 1.0 (linear)?

Exercise 4.5 (Programming – Basic). Use gensim to load pre-trained Word2Vec vectors (e.g., `word2vec-google-news-300`). Query `most_similar("python")` and observe the results. Then query `most_similar(positive=["python", "programming"])` and `most_similar(positive=["python", "snake"])`. Does the static embedding capture both senses of "python"? Discuss the limitation in terms of the polysemy problem from Section 4.5.3.

Exercise 4.6 (Programming – Intermediate). Train Word2Vec (using gensim) on a corpus of your choice with embedding dimensions $d \in \{50, 100, 150, 200, 250, 300\}$, keeping all other hyperparameters fixed. For each dimension, evaluate on the Google analogy test using `wv.evaluate_word_analogies()`. Plot accuracy versus dimension and discuss: (a) at what dimension does accuracy plateau, and (b) what does this suggest about the intrinsic dimensionality of word semantics?

Exercise 4.7 (Programming – Intermediate). Construct a word-context co-occurrence matrix from a corpus of at least 1,000 sentences (you may use NLTK's `brown` corpus or similar), with a window of $k = 2$. Apply truncated SVD with $k = 50$ components using `sklearn.decomposition.TruncatedSVD`. For five word pairs of your choice, compare the cosine similarity from your SVD vectors to the cosine similarity from pre-trained Word2Vec vectors. How close are the two methods?

Hint: Use `scipy.sparse.lil_matrix` for efficient sparse matrix construction, and normalize rows before computing cosine similarity.

Exercise 4.8 (Programming – Intermediate). Load pre-trained FastText vectors via gensim. Test the model on five out-of-vocabulary compound words (e.g., "unbreathable," "oversnowified," "microplasticize"). For each OOV word, print the five most similar known words. Then attempt the same with Word2Vec and report the error. Discuss why FastText can handle these words while Word2Vec cannot.

Exercise 4.9 (Programming – Advanced). Investigate gender bias in word embeddings. Load pre-trained Word2Vec vectors and compute the vector $\mathbf{v}_{\text{he}} - \mathbf{v}_{\text{she}}$. Find the 10 words most aligned with this "gender direction" (highest cosine similarity to the difference vector) and the 10 words

most opposed to it. Report your results and discuss whether the patterns reflect societal stereotypes. Propose (but do not necessarily implement) a debiasing strategy based on Bolukbasi et al. [2016].

Exercise 4.10 (Programming – Advanced). Implement Skip-gram with negative sampling from scratch using NumPy (not PyTorch or TensorFlow). Use a vocabulary of 500 words from a small corpus. Your implementation should:

(a) Build a vocabulary and training pairs from the corpus with window size $c = 2$.

(b) Initialize two embedding matrices (input and output) of dimension $d = 50$.

(c) For each positive pair, sample $K = 5$ negative words from the unigram distribution raised to the 3/4 power.

(d) Compute the loss from Equation 4.3 and update embeddings using SGD with learning rate $\eta = 0.01$.

(e) After 10 epochs, query the 5 nearest neighbors for 3 words of your choice.

Compare your embeddings to gensim's Word2Vec trained on the same corpus. How similar are the nearest-neighbor lists?