

-->

Chapter 4: Representation Learning — Vector Semantics and Neural Networks

Covered Material	Value
Chapters	Chapters 6 and 7 (Jurafsky & Martin)
Related Week	Week 4
Related Slides / Files	week4.md , 6.md , 7.md

Index

- Index (this list)
- Brief Content Summary
- Abstract
- Keywords
- 1 Introduction
 - 1.1 Motivation for Representation Learning
 - 1.2 Chapter Overview
- 2 Lexical Semantics
 - 2.1 Words and Word Senses
 - 2.2 Synonymy, Similarity, and Relatedness
 - 2.3 The Distributional Hypothesis
- 3 Vector Space Models
 - 3.1 Term-Document Matrices
 - 3.2 Term-Term (Word-Word) Matrices
 - 3.3 Weighting Schemes: TF-IDF
- 4 Measuring Semantic Similarity
 - 4.1 Cosine Similarity
 - 4.2 Properties of Cosine Distance
- 5 Pointwise Mutual Information
 - 5.1 PMI Definition and Intuition

- 5.2 Positive PMI (PPMI)
 - 5.3 Worked Example: Computing PPMI
- 6 Word Embeddings
 - 6.1 Dense Vector Representations
 - 6.2 Word2Vec and Skip-Gram
 - 6.3 Skip-Gram with Negative Sampling (SGNS)
 - 6.4 Learning the Embeddings
 - 6.5 Properties of Learned Embeddings
 - 6.6 Bias in Embeddings
- 7 Neural Network Fundamentals
 - 7.1 The Neural Unit
 - 7.2 Activation Functions
 - 7.3 The XOR Problem and Multi-Layer Networks
- 8 Feedforward Neural Networks
 - 8.1 Architecture
 - 8.2 The Hidden Layer
 - 8.3 The Output Layer and Softmax
 - 8.4 Classification with Neural Networks
- 9 Training Neural Networks
 - 9.1 Loss Functions: Cross-Entropy
 - 9.2 Gradient Descent
 - 9.3 Computation Graphs
 - 9.4 Backpropagation
 - 9.5 Practical Considerations
- 10 Neural Language Models
 - 10.1 Feedforward Language Models
 - 10.2 Forward Inference
 - 10.3 Training Neural LMs
- 11 Conclusion
- 12 References
- Appendix A: Mathematical Derivations

- Glossary

Brief Content Summary

This chapter addresses the fundamental question of how to represent meaning computationally. Beginning with lexical semantics and the distributional hypothesis ("you shall know a word by the company it keeps"), the chapter develops vector space models for representing words as high-dimensional vectors based on co-occurrence patterns. Central weighting schemes including Term Frequency-Inverse Document Frequency (TF-IDF) and Positive Pointwise Mutual Information (PPMI) are derived and illustrated. The chapter then transitions to dense, learned representations—word embeddings—focusing on the Word2Vec Skip-Gram model with Negative Sampling (SGNS). The second half introduces neural network fundamentals: the computational unit, activation functions (sigmoid, tanh, ReLU), and multi-layer feedforward architectures. Training via cross-entropy loss, gradient descent, and backpropagation through computation graphs is formally presented. The chapter concludes with feedforward neural language models, demonstrating how these components integrate into systems that predict upcoming words from prior context.

Abstract

This chapter provides a rigorous treatment of representation learning in Natural Language Processing (NLP), spanning both count-based vector semantics and neural network-based approaches. The exposition begins with lexical semantics, establishing foundational notions of word meaning, synonymy, similarity, and the distributional hypothesis that underpins modern distributional semantics. Vector space models are constructed via term-document and term-term matrices, with formal derivations of TF-IDF weighting and cosine similarity as the primary similarity metric. Pointwise Mutual Information (PMI) and its positive variant (PPMI) are introduced as association measures superior to raw counts. The chapter then develops Word2Vec's Skip-Gram architecture, formulating embedding learning as a binary classification task that distinguishes true context words from noise samples. The transition to neural networks covers computational units with nonlinear activations, the XOR problem as motivation for hidden layers, and feedforward network architectures for text classification. Training procedures including cross-entropy loss, stochastic gradient descent, and error backpropagation via computation graphs are presented with mathematical precision. The chapter culminates with feedforward neural language models that learn to predict the next word while simultaneously acquiring useful word representations.

Keywords

Vector Semantics; Distributional Hypothesis; TF-IDF; Cosine Similarity; Pointwise Mutual Information; PPMI; Word Embeddings; Word2Vec; Skip-Gram; Negative Sampling; Neural Networks; Activation Functions; Feedforward Networks; Softmax; Cross-Entropy Loss; Backpropagation; Computation Graphs; Neural Language Models

1 Introduction

This section establishes the motivation for learning vector representations of words and provides an overview of the chapter's structure.

1.1 Motivation for Representation Learning

A fundamental challenge in Natural Language Processing is determining how to represent the meaning of words in a form amenable to computational processing. Traditional symbolic approaches, where words are represented as atomic symbols with no inherent structure, suffer from a critical limitation: they cannot capture the relationships between words. For instance, such representations provide no mechanism to determine that "cat" and "dog" are more similar to each other than either is to "democracy."

The solution developed in this chapter relies on the insight that word meaning can be inferred from usage patterns. Words that appear in similar contexts tend to have similar meanings—a principle known as the **distributional hypothesis**. This insight motivates the construction of vector representations where semantic relationships manifest as geometric relationships in a vector space.

Two paradigms for constructing such representations are explored:

1. **Count-based methods:** Words are represented by vectors of co-occurrence counts (or weighted transformations thereof) with other words or documents. These methods are transparent and interpretable but yield sparse, high-dimensional vectors.
2. **Prediction-based methods:** Dense, low-dimensional vectors (embeddings) are learned by training neural networks to predict words from their contexts. These methods produce representations that capture subtle semantic and syntactic regularities.

The second half of the chapter introduces the neural network machinery required to understand prediction-based embeddings and, more broadly, modern deep learning approaches to NLP.

1.2 Chapter Overview

The chapter proceeds as follows:

- **Sections 2–4** develop count-based vector semantics, covering lexical semantics foundations, vector space construction via term-document and term-term matrices, TF-IDF weighting, and cosine similarity.
 - **Section 5** introduces Pointwise Mutual Information (PMI) as an association measure that addresses limitations of raw co-occurrence counts.
 - **Section 6** transitions to dense embeddings, presenting the Word2Vec Skip-Gram model and its training via negative sampling.
 - **Sections 7–9** provide a self-contained introduction to neural networks, covering computational units, activation functions, feedforward architectures, and training via backpropagation.
 - **Section 10** synthesizes the preceding material in the context of feedforward neural language models.
-

2 Lexical Semantics

This section introduces foundational concepts from lexical semantics that inform the computational treatment of word meaning.

2.1 Words and Word Senses

The study of word meaning requires distinguishing between several related concepts:

Lemmas and Wordforms. A **lemma** is a canonical form representing a set of inflected wordforms. For example, the lemma *sing* encompasses the wordforms *sing*, *sang*, *sung*, and *singing*. Lemmas are typically what appear as headwords in dictionaries.

Word Senses. A single lemma may have multiple distinct meanings, each called a **sense**. The word *bank* has at least two senses:

- A financial institution
- The sloping land beside a body of water

Words with multiple senses are termed **polysemous**. The phenomenon of polysemy presents challenges for computational models, as a single wordform may require different representations depending on context.

2.2 Synonymy, Similarity, and Relatedness

Several distinct relationships hold between word meanings:

Synonymy. Two words are **synonyms** if they have the same meaning in some or all contexts. Perfect synonymy is rare; most synonyms differ in nuance, register, or collocational preferences. For example, *big* and *large* are near-synonyms but exhibit different usage patterns (*big sister* vs. *?large sister*).

The formal criterion for synonymy is **substitutability**: two words are synonymous if substituting one for the other preserves the truth conditions of any sentence.

Similarity. Words may be similar without being synonymous. Similarity is often defined via shared features or taxonomic proximity. For instance, *cat* and *dog* are similar (both are mammals, pets, quadrupeds) but are not synonyms.

Relatedness (Association). A broader notion captures words that co-occur or are thematically connected without necessarily being similar. *Coffee* and *cup* are related but not similar—they belong to different semantic categories but frequently co-occur.

These distinctions matter because different vector space models capture different aspects of meaning. Co-occurrence-based models often capture relatedness (association), while taxonomic similarity requires additional structure.

2.3 The Distributional Hypothesis

The theoretical foundation for vector semantics is the **distributional hypothesis**, articulated by linguists including Zellig Harris (1954) and John Firth (1957). Firth's formulation is often quoted:

"You shall know a word by the company it keeps."

The hypothesis asserts that words occurring in similar linguistic contexts tend to have similar meanings. This principle enables the inference of semantic properties from observable distributional patterns, without requiring access to referents or world knowledge.

Formally, if w_1 and w_2 appear with overlapping sets of context words, we infer that w_1 and w_2 are semantically related. The strength of this inference scales with the degree of contextual overlap.

3 Vector Space Models

This section develops the construction of vector representations from co-occurrence statistics.

3.1 Term-Document Matrices

The simplest vector space model represents words based on their occurrence in documents. Given a vocabulary V and a collection of D documents, the **term-document matrix** \mathbf{M} has dimensions $|V| \times D$, where entry M_{ij} records the frequency of word i in document j .

Each row of \mathbf{M} is a vector representation of a word, where dimensions correspond to documents. Words appearing in similar documents will have similar row vectors.

Example. Consider four words across four Shakespeare plays:

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
soldier	2	0	12	36
fool	37	58	1	5
crown	5	117	0	0

The vectors for *battle* and *soldier* are similar (high values for the history plays), while *fool* has a different distribution (concentrated in the comedies).

Limitations. Term-document matrices conflate topical similarity with semantic similarity. Documents on related topics will induce similarity between words that co-occur with those topics, even if the words themselves are not semantically related.

3.2 Term-Term (Word-Word) Matrices

A more direct approach to capturing word meaning constructs vectors based on co-occurrence with other words, yielding a **term-term matrix** (also called a **word-word matrix** or **word-context matrix**).

Given vocabulary V , the term-term matrix \mathbf{M} has dimensions $|V| \times |V|$, where entry M_{ij} records how often word i occurs in the context of word j . Context is typically defined by a window of $\pm k$ words around the target.

Each row is now a vector of length $|V|$, where dimensions correspond to context words. Words with similar co-occurrence patterns will have similar vectors.

Example. With a context window of ± 7 words in a corpus:

	aardvark	computer	data	pinch	result	sugar
apricot	0	0	0	1	0	1
pineapple	0	0	0	1	0	1
digital	0	2	1	0	1	0
information	0	1	6	0	4	0

The vectors for *apricot* and *pineapple* are identical, reflecting their similar usage in contexts involving food preparation (*pinch*, *sugar*). The vectors for *digital* and *information* are similar and distinct from the fruit words.

3.3 Weighting Schemes: TF-IDF

Raw co-occurrence counts are problematic because frequent words (e.g., *the*, *of*) dominate the counts without contributing discriminative information. **Term Frequency-Inverse Document Frequency (TF-IDF)** weighting addresses this by upweighting informative terms and downweighting common terms.

Term Frequency (TF)

Raw term frequency $\text{tf}_{t,d}$ is the count of term t in document d . To reduce the impact of very high frequencies, a log transformation is applied:

$$\text{tf}_{t,d} = \begin{cases} 1 + \log_{10}(\text{count}(t, d)) & \text{if } \text{count}(t, d) > 0 \\ 0 & \text{otherwise} \end{cases} \quad (3.1)$$

Inverse Document Frequency (IDF)

Terms appearing in many documents are less informative for distinguishing documents. Let df_t denote the **document frequency** of term t (the number of documents containing t), and let N be the total number of documents. The inverse document frequency is:

$$\text{idf}_t = \log_{10} \left(\frac{N}{\text{df}_t} \right) \quad (3.2)$$

Rare terms have high IDF; terms appearing in all documents have $\text{idf} = 0$.

Combined TF-IDF Weight

The TF-IDF weight combines both factors:

$$w_{t,d} = \text{tf}_{t,d} \times \text{idf}_t \quad (3.3)$$

This weighting scheme is foundational in information retrieval and serves as a preprocessing step for many NLP tasks.

4 Measuring Semantic Similarity

This section develops cosine similarity as the standard metric for comparing word vectors.

4.1 Cosine Similarity

Given two word vectors, we require a measure of their similarity. The **dot product** is a natural candidate:

$$\mathbf{v} \cdot \mathbf{w} = \sum_{i=1}^N v_i w_i \quad (4.1)$$

However, the dot product is unbounded and scales with vector magnitude—longer vectors yield larger products regardless of directional similarity.

Cosine similarity normalizes by vector lengths, measuring the cosine of the angle between vectors:

$$\text{cosine}(\mathbf{v}, \mathbf{w}) = \frac{\mathbf{v} \cdot \mathbf{w}}{|\mathbf{v}| |\mathbf{w}|} = \frac{\sum_{i=1}^N v_i w_i}{\sqrt{\sum_{i=1}^N v_i^2} \sqrt{\sum_{i=1}^N w_i^2}} \quad (4.2)$$

where $|\mathbf{v}| = \sqrt{\sum_i v_i^2}$ is the Euclidean (ℓ_2) norm.

Cosine similarity ranges from -1 to $+1$:

- **cosine** = 1: vectors point in the same direction (maximally similar)
- **cosine** = 0: vectors are orthogonal (no similarity)
- **cosine** = -1 : vectors point in opposite directions

For non-negative vectors (e.g., count-based representations), cosine similarity ranges from 0 to 1.

Worked Example. Consider computing the similarity between *apricot* and *digital* using the term-term matrix from Section 3.2:

$$\mathbf{v}_{\text{apricot}} = [0, 0, 0, 1, 0, 1]$$

$$\mathbf{v}_{\text{digital}} = [0, 2, 1, 0, 1, 0]$$

Dot product:

$$\mathbf{v}_{\text{apricot}} \cdot \mathbf{v}_{\text{digital}} = (0)(0) + (0)(2) + (0)(1) + (1)(0) + (0)(1) + (1)(0) = 0$$

The vectors are orthogonal—the words share no context words and are thus unrelated according to this representation.

Now compare *apricot* and *pineapple*:

$$\mathbf{v}_{\text{pineapple}} = [0, 0, 0, 1, 0, 1]$$

$$\text{cosine}(\mathbf{v}_{\text{apricot}}, \mathbf{v}_{\text{pineapple}}) = \frac{0 + 0 + 0 + 1 + 0 + 1}{\sqrt{2} \cdot \sqrt{2}} = \frac{2}{2} = 1$$

The vectors are identical, yielding maximal similarity.

4.2 Properties of Cosine Distance

The **cosine distance** is defined as:

$$d_{\text{cosine}}(\mathbf{v}, \mathbf{w}) = 1 - \text{cosine}(\mathbf{v}, \mathbf{w}) \quad (4.3)$$

This transforms similarity to distance (dissimilarity).

Cosine similarity is preferable to Euclidean distance for word vectors because:

1. **Scale invariance:** Cosine similarity is unaffected by vector magnitude. A word appearing twice as often in a corpus does not become "more similar" to itself.
2. **Sparsity handling:** In high-dimensional sparse vectors, most dimensions are zero. Euclidean distance is dominated by these zeros, while cosine focuses on the dimensions where both vectors have non-zero values.
3. **Interpretability:** The angular interpretation provides geometric intuition—similar words cluster in similar directions from the origin.

5 Pointwise Mutual Information

This section introduces PMI and PPMI as association measures that address limitations of raw co-occurrence counts.

5.1 PMI Definition and Intuition

Raw co-occurrence counts are biased toward frequent words. The word *the* co-occurs frequently with nearly every word, not because of semantic affinity but because of its high frequency. We seek a measure that asks: "Does this word co-occur with this context more than we would expect by chance?"

Pointwise Mutual Information (PMI) measures the discrepancy between the observed co-occurrence probability and the probability expected under independence:

$$\text{PMI}(w, c) = \log_2 \frac{P(w, c)}{P(w)P(c)} \quad (5.1)$$

where:

- $P(w, c)$ is the probability of observing word w and context word c together
- $P(w)$ is the marginal probability of w

- $P(c)$ is the marginal probability of c

Interpretation:

- $\text{PMI} > 0$: The words co-occur more often than expected under independence (positive association)
- $\text{PMI} = 0$: Co-occurrence matches the independence expectation
- $\text{PMI} < 0$: The words co-occur less often than expected (negative association, or avoidance)

5.2 Positive PMI (PPMI)

Negative PMI values are problematic:

1. **Unreliability**: Reliably estimating that two words avoid each other requires observing their *absence* of co-occurrence across a large corpus. Such estimates are noisy.
2. **Sparsity**: Most word pairs never co-occur, yielding $\text{PMI} = -\infty$ (since $\log 0 = -\infty$).

The standard solution is **Positive PMI (PPMI)**, which clamps negative values to zero:

$$\text{PPMI}(w, c) = \max \left(\log_2 \frac{P(w, c)}{P(w)P(c)}, 0 \right) \quad (5.2)$$

PPMI matrices are sparse and non-negative, making them suitable for cosine similarity computations.

5.3 Worked Example: Computing PPMI

Consider a small corpus yielding the following co-occurrence matrix with a context window of ± 1 word:

	computer	data	result	pie	sugar
cherry	0	0	0	5	10
strawberry	0	0	0	3	8
digital	10	20	5	0	0
information	15	25	10	0	0

Step 1: Compute probabilities.

Total count: $N = 0 + 0 + 0 + 5 + 10 + 0 + 0 + 0 + 3 + 8 + 10 + 20 + 5 + 0 + 0 + 15 + 25 + 10 + 0 + 0 = 111$

Marginal probabilities for words (rows):

$$P(\text{cherry}) = \frac{15}{111}, \quad P(\text{strawberry}) = \frac{11}{111}, \quad P(\text{digital}) = \frac{35}{111}, \quad P(\text{information}) = \frac{50}{111}$$

Marginal probabilities for contexts (columns):

$$P(\text{computer}) = \frac{25}{111}, \quad P(\text{data}) = \frac{45}{111}, \quad P(\text{result}) = \frac{15}{111}$$

$$P(\text{pie}) = \frac{8}{111}, \quad P(\text{sugar}) = \frac{18}{111}$$

Step 2: Compute PMI for selected pairs.

For (cherry, sugar):

$$P(\text{cherry, sugar}) = \frac{10}{111}$$

$$\text{PMI}(\text{cherry, sugar}) = \log_2 \frac{10/111}{(15/111)(18/111)} = \log_2 \frac{10 \cdot 111}{15 \cdot 18} = \log_2 \frac{1110}{270} = \log_2 4.11 \approx 2.04$$

For (digital, sugar):

$$P(\text{digital, sugar}) = \frac{0}{111} = 0$$

$$\text{PMI}(\text{digital, sugar}) = \log_2 0 = -\infty$$

Thus $\text{PPMI}(\text{digital, sugar}) = 0$.

For (information, data):

$$P(\text{information, data}) = \frac{25}{111}$$

$$\text{PMI}(\text{information, data}) = \log_2 \frac{25/111}{(50/111)(45/111)} = \log_2 \frac{25 \cdot 111}{50 \cdot 45} = \log_2 \frac{2775}{2250} = \log_2 1.23 \approx 0.30$$

Observation: PPMI assigns high values to word-context pairs that co-occur more than expected (cherry-sugar: 2.04), moderate values to expected associations (information-data: 0.30), and zero to pairs that never co-occur (digital-sugar: 0).

6 Word Embeddings

This section transitions from sparse, count-based representations to dense, learned embeddings.

6.1 Dense Vector Representations

PPMI matrices, while effective, have significant drawbacks:

1. **High dimensionality:** Vectors have $|V|$ dimensions, where vocabulary sizes can reach hundreds of thousands.
2. **Sparsity:** Most entries are zero, wasting storage and computational resources.
3. **Limited generalization:** The representation cannot leverage similarity between context words.

Word embeddings address these issues by representing words as dense vectors of much lower dimensionality (typically 50–300 dimensions). These representations are learned from data such that:

- Similar words receive similar embeddings
- Semantic and syntactic regularities emerge as algebraic relationships

The term "embedding" reflects that words are embedded as points in a continuous vector space.

6.2 Word2Vec and Skip-Gram

Word2Vec (Mikolov et al., 2013) is a family of algorithms for learning word embeddings. The **Skip-Gram** model, the more widely used variant, learns embeddings by training a classifier to predict context words given a target word.

The key insight is that rather than counting co-occurrences explicitly, we train a model whose parameters encode the same distributional information implicitly. The trained parameters become the word embeddings.

Skip-Gram Objective. Given a target word w and a context word c , the Skip-Gram model learns to estimate:

$$P(c | w) \quad (6.1)$$

For a window of $\pm L$ words, the model predicts each context word independently:

$$P(c_{-L}, \dots, c_{-1}, c_{+1}, \dots, c_{+L} | w) = \prod_{-L \leq j \leq L, j \neq 0} P(c_j | w) \quad (6.2)$$

Representation. Each word w is associated with two vectors:

- **Target embedding** $\mathbf{w} \in \mathbb{R}^d$: used when w is the target word
- **Context embedding** $\mathbf{c} \in \mathbb{R}^d$: used when w is a context word

The probability is modeled using the sigmoid of the dot product:

$$P(+ | w, c) = \sigma(\mathbf{c} \cdot \mathbf{w}) = \frac{1}{1 + e^{-\mathbf{c} \cdot \mathbf{w}}} \quad (6.3)$$

where $P(+ | w, c)$ is the probability that c is a true context word of w .

6.3 Skip-Gram with Negative Sampling (SGNS)

Training a classifier to distinguish true context words from all other words is computationally prohibitive (softmax over $|V|$ classes). **Negative sampling** transforms this into a tractable binary classification problem.

Training Setup:

1. **Positive examples:** (w, c) pairs from the actual corpus, labeled $+$
2. **Negative examples:** (w, c_{neg}) pairs where c_{neg} is sampled randomly, labeled $-$

For each positive pair, we sample k negative examples (typically $k = 5-15$).

Loss Function. For a single training instance with target w , positive context c_{pos} , and k negative samples $c_{\text{neg}_1}, \dots, c_{\text{neg}_k}$:

$$L = - \left[\log \sigma(\mathbf{c}_{\text{pos}} \cdot \mathbf{w}) + \sum_{i=1}^k \log \sigma(-\mathbf{c}_{\text{neg}_i} \cdot \mathbf{w}) \right] \quad (6.4)$$

Minimizing this loss:

- Increases the dot product $\mathbf{c}_{\text{pos}} \cdot \mathbf{w}$ (making true pairs similar)
- Decreases the dot product $\mathbf{c}_{\text{neg}} \cdot \mathbf{w}$ (making noise pairs dissimilar)

Negative Sampling Distribution. Negative samples are drawn according to:

$$P_{\alpha}(w) = \frac{\text{count}(w)^{\alpha}}{\sum_{w'} \text{count}(w')^{\alpha}} \quad (6.5)$$

where $\alpha = 0.75$ is typical. This raises the probability of rare words relative to uniform sampling while still favoring frequent words.

6.4 Learning the Embeddings

The embeddings are learned via stochastic gradient descent. Given the loss in Equation 6.4, we compute gradients with respect to both \mathbf{w} and \mathbf{c} .

Gradient for target embedding:

$$\frac{\partial L}{\partial \mathbf{w}} = [\sigma(\mathbf{c}_{\text{pos}} \cdot \mathbf{w}) - 1] \mathbf{c}_{\text{pos}} + \sum_{i=1}^k \sigma(\mathbf{c}_{\text{neg}_i} \cdot \mathbf{w}) \mathbf{c}_{\text{neg}_i} \quad (6.6)$$

Gradient for context embedding:

$$\frac{\partial L}{\partial \mathbf{c}_{\text{pos}}} = [\sigma(\mathbf{c}_{\text{pos}} \cdot \mathbf{w}) - 1] \mathbf{w} \quad (6.7)$$

$$\frac{\partial L}{\partial \mathbf{c}_{\text{neg}_i}} = \sigma(\mathbf{c}_{\text{neg}_i} \cdot \mathbf{w}) \mathbf{w} \quad (6.8)$$

Update rules (with learning rate η):

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \frac{\partial L}{\partial \mathbf{w}} \quad (6.9)$$

$$\mathbf{c}^{(t+1)} = \mathbf{c}^{(t)} - \eta \frac{\partial L}{\partial \mathbf{c}} \quad (6.10)$$

After training, the target embedding matrix \mathbf{W} is typically used as the word embedding matrix. Alternatively, \mathbf{W} and \mathbf{C} can be summed or concatenated.

6.5 Properties of Learned Embeddings

Word2Vec embeddings exhibit remarkable properties:

Semantic Clustering. Words with similar meanings cluster together in the embedding space. Nearest neighbors of "France" include "Spain," "Belgium," and "Germany."

Analogical Reasoning. Embeddings capture relational structure such that:

$$\mathbf{v}_{\text{king}} - \mathbf{v}_{\text{man}} + \mathbf{v}_{\text{woman}} \approx \mathbf{v}_{\text{queen}} \quad (6.11)$$

This suggests that the vector offset $\mathbf{v}_{\text{king}} - \mathbf{v}_{\text{man}}$ encodes a gender-neutral "royalty" concept.

Parallelogram Structure. Analogies manifest geometrically as parallelograms:

$$\mathbf{v}_{\text{king}} - \mathbf{v}_{\text{queen}} \approx \mathbf{v}_{\text{man}} - \mathbf{v}_{\text{woman}} \quad (6.12)$$

These properties emerge from the training objective without explicit supervision for analogies.

6.6 Bias in Embeddings

Because embeddings are learned from text corpora, they inherit biases present in those corpora. Studies have documented:

- **Gender stereotypes:** The vector $\mathbf{v}_{\text{doctor}} - \mathbf{v}_{\text{man}} + \mathbf{v}_{\text{woman}}$ is closer to "nurse" than to "doctor"
- **Racial biases:** African American names cluster with negative sentiment words
- **Occupational biases:** Female-associated terms cluster with domestic roles

These biases can propagate to downstream applications (e.g., resume screening, sentiment analysis). Research on **debiasing** embeddings seeks to identify and mitigate these effects while preserving useful semantic information.

Implications. The presence of bias in embeddings is a reminder that distributional semantics captures usage patterns, not normative meaning. Models trained on biased text will learn biased representations.

7 Neural Network Fundamentals

This section introduces the building blocks of neural networks.

7.1 The Neural Unit

The fundamental computational element is the **neural unit** (also called a **node** or **neuron**). A neural unit computes a weighted sum of its inputs, adds a bias term, and applies a nonlinear activation function.

Given an input vector $\mathbf{x} = [x_1, x_2, \dots, x_n]^T$, weight vector $\mathbf{w} = [w_1, w_2, \dots, w_n]^T$, and bias b , the unit computes:

$$z = \mathbf{w} \cdot \mathbf{x} + b = \sum_{i=1}^n w_i x_i + b \quad (7.1)$$

$$y = f(z) \quad (7.2)$$

where f is the **activation function** and y is the unit's output.

The bias b allows the unit to shift its activation threshold. Without bias, the decision boundary must pass through the origin.

7.2 Activation Functions

The choice of activation function determines the unit's nonlinear behavior.

Sigmoid. The sigmoid function squashes real values to the range $(0, 1)$:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad (7.3)$$

Properties:

- $\sigma(z) \rightarrow 1$ as $z \rightarrow +\infty$
- $\sigma(z) \rightarrow 0$ as $z \rightarrow -\infty$
- $\sigma(0) = 0.5$
- Derivative: $\frac{d\sigma}{dz} = \sigma(z)(1 - \sigma(z))$

Sigmoid is used for binary classification outputs but suffers from the **vanishing gradient problem**: gradients approach zero for large $|z|$, slowing learning.

Hyperbolic Tangent (tanh). Tanh maps inputs to the range $(-1, 1)$:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \quad (7.4)$$

Tanh is zero-centered, which can improve gradient flow. It is related to sigmoid by:

$$\tanh(z) = 2\sigma(2z) - 1 \quad (7.5)$$

Derivative: $\frac{d \tanh}{dz} = 1 - \tanh^2(z)$

Rectified Linear Unit (ReLU). ReLU is the most widely used activation in modern deep learning:

$$\text{ReLU}(z) = \max(0, z) \quad (7.6)$$

Derivative:

$$\frac{d\text{ReLU}}{dz} = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \quad (7.7)$$

ReLU advantages:

- Computationally efficient (no exponentials)
- Constant gradient for positive inputs (mitigates vanishing gradients)
- Induces sparsity (negative inputs produce zero output)

ReLU disadvantage: "Dead neurons" can occur when units get stuck at zero output.

7.3 The XOR Problem and Multi-Layer Networks

A single neural unit implements a linear classifier, which cannot represent nonlinearly separable functions. The canonical example is **XOR** (exclusive or):

x_1	x_2	$x_1 \text{ XOR } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

No single line can separate the $+$ class (where output is 1) from the $-$ class. XOR is not linearly separable.

Solution: Multi-Layer Networks. Adding a **hidden layer** between input and output creates a two-layer network that can represent XOR.

Consider a network with:

- Input layer: x_1, x_2
- Hidden layer: 2 units with ReLU activation
- Output layer: 1 unit with sigmoid activation

One valid solution:

Hidden unit 1: $h_1 = \max(0, x_1 + x_2 - 0.5)$

Hidden unit 2: $h_2 = \max(0, -x_1 - x_2 + 1.5)$

Output: $y = \sigma(h_1 + h_2 - 1)$

The hidden layer transforms the input space such that the output becomes linearly separable.

Universal Approximation. A feedforward network with a single hidden layer containing enough units can approximate any continuous function to arbitrary precision (Cybenko, 1989). This theoretical result motivates the use of hidden layers for modeling complex functions.

8 Feedforward Neural Networks

This section develops the architecture and operation of feedforward networks.

8.1 Architecture

A **feedforward neural network** (also called a **multi-layer perceptron** or MLP) consists of:

1. **Input layer:** Receives the feature vector $\mathbf{x} \in \mathbb{R}^{n_0}$

2. **Hidden layers:** One or more layers of processing units
3. **Output layer:** Produces the final prediction

In a fully connected (dense) network, every unit in layer l connects to every unit in layer $l + 1$. Information flows forward from input to output with no cycles.

Notation. For a network with L layers:

- $\mathbf{W}^{[l]}$: weight matrix for layer l (dimensions $n_l \times n_{l-1}$)
- $\mathbf{b}^{[l]}$: bias vector for layer l (dimensions $n_l \times 1$)
- $\mathbf{a}^{[l]}$: activation vector for layer l
- $\mathbf{z}^{[l]}$: pre-activation (weighted sum) for layer l

8.2 The Hidden Layer

The hidden layer transforms the input representation. For a single hidden layer:

$$\mathbf{z}^{[1]} = \mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]} \quad (8.1)$$

$$\mathbf{h} = g(\mathbf{z}^{[1]}) \quad (8.2)$$

where g is the activation function applied element-wise.

The hidden layer learns an **internal representation** of the input that is useful for the task. These learned representations are the "deep" features that give deep learning its power.

Dimensionality. If the input has n_0 dimensions and the hidden layer has n_1 units:

- $\mathbf{W}^{[1]} \in \mathbb{R}^{n_1 \times n_0}$
- $\mathbf{b}^{[1]} \in \mathbb{R}^{n_1}$
- $\mathbf{h} \in \mathbb{R}^{n_1}$

The hidden layer can expand ($n_1 > n_0$), compress ($n_1 < n_0$), or preserve dimensionality.

8.3 The Output Layer and Softmax

For classification tasks, the output layer produces a probability distribution over K classes.

Binary Classification. A single output unit with sigmoid activation:

$$\hat{y} = \sigma(\mathbf{w} \cdot \mathbf{h} + b) \quad (8.3)$$

where $\hat{y} \in (0, 1)$ is interpreted as $P(y = 1|\mathbf{x})$.

Multi-class Classification. The **softmax** function generalizes sigmoid to K classes:

$$\mathbf{z} = \mathbf{U}\mathbf{h} + \mathbf{b}^{[2]} \quad (8.4)$$

$$\hat{y}_k = \text{softmax}(z_k) = \frac{\exp(z_k)}{\sum_{j=1}^K \exp(z_j)} \quad (8.5)$$

Softmax properties:

- Outputs sum to 1: $\sum_k \hat{y}_k = 1$
- All outputs are positive: $\hat{y}_k > 0$
- Exponential amplifies differences: larger z_k receive disproportionately higher probability

The output $\hat{\mathbf{y}} = [\hat{y}_1, \dots, \hat{y}_K]^T$ is a probability distribution over classes.

8.4 Classification with Neural Networks

Complete Forward Pass. For a 2-layer network (1 hidden layer):

$$\mathbf{h} = g(\mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]}) \quad (8.6)$$

$$\mathbf{z} = \mathbf{W}^{[2]}\mathbf{h} + \mathbf{b}^{[2]} \quad (8.7)$$

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z}) \quad (8.8)$$

Using Pre-trained Embeddings. For text classification, the input \mathbf{x} is often derived from word embeddings. Common strategies:

1. **Embedding lookup:** For single words, retrieve the embedding vector
2. **Embedding average:** Average the embeddings of all words in a document
3. **Concatenation:** For fixed-length sequences, concatenate embeddings

Example: Sentiment Classification. Given a sentence, we:

1. Look up embeddings for each word
2. Average the embeddings to get \mathbf{x}
3. Pass through hidden layer: $\mathbf{h} = \text{ReLU}(\mathbf{W}\mathbf{x} + \mathbf{b})$
4. Compute output: $\hat{y} = \text{softmax}(\mathbf{U}\mathbf{h})$

The network learns weights that project the embedding space into a sentiment-predictive representation.

9 Training Neural Networks

This section presents the theory and algorithms for training neural networks.

9.1 Loss Functions: Cross-Entropy

Training requires a **loss function** that quantifies the discrepancy between predictions and ground truth. For classification, the standard choice is **cross-entropy loss**.

Binary Cross-Entropy. For binary classification with true label $y \in \{0, 1\}$ and predicted probability $\hat{y} = P(y = 1 | \mathbf{x})$:

$$L_{\text{CE}}(\hat{y}, y) = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})] \quad (9.1)$$

This loss:

- Equals $-\log \hat{y}$ when $y = 1$ (penalizes low confidence in positive class)
- Equals $-\log(1 - \hat{y})$ when $y = 0$ (penalizes high confidence in negative class)

Multi-class Cross-Entropy. For K classes with true label vector \mathbf{y} (one-hot encoded) and predicted probabilities $\hat{\mathbf{y}}$:

$$L_{\text{CE}}(\hat{\mathbf{y}}, \mathbf{y}) = - \sum_{k=1}^K y_k \log \hat{y}_k \quad (9.2)$$

Since \mathbf{y} is one-hot (only one $y_k = 1$), this simplifies to:

$$L_{\text{CE}} = -\log \hat{y}_c \quad (9.3)$$

where c is the index of the correct class. Minimizing cross-entropy is equivalent to maximizing the log probability of the correct class.

9.2 Gradient Descent

Neural networks are trained by minimizing the loss function using **gradient descent**. The gradient indicates the direction of steepest increase; we move in the opposite direction.

Update Rule. Given parameters θ and learning rate η :

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_{\theta} L \quad (9.4)$$

where $\nabla_{\theta} L$ is the gradient of the loss with respect to parameters.

Stochastic Gradient Descent (SGD). Computing the gradient over the entire training set is expensive. SGD approximates the full gradient using a single example (or mini-batch):

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_{\theta} L(\mathbf{x}^{(i)}, y^{(i)}) \quad (9.5)$$

SGD is noisy but computationally efficient and often escapes local minima.

Mini-batch Gradient Descent. A compromise uses batches of B examples:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \frac{1}{B} \sum_{i=1}^B \nabla_{\theta} L(\mathbf{x}^{(i)}, y^{(i)}) \quad (9.6)$$

Typical batch sizes range from 16 to 256.

9.3 Computation Graphs

A **computation graph** represents the forward computation as a directed acyclic graph (DAG) where:

- Nodes represent variables (inputs, intermediate values, outputs)
- Edges represent operations

Computation graphs enable systematic gradient computation via the chain rule.

Example. Consider $L(a, b, c) = c \cdot (a + 2b)$.

Decompose into elementary operations:

- $d = 2b$
- $e = a + d$
- $L = c \cdot e$

The graph flows from inputs (a, b, c) through intermediate nodes (d, e) to output L .

For $a = 3, b = 1, c = -2$:

- $d = 2$
- $e = 5$
- $L = -10$

9.4 Backpropagation

Backpropagation (backward differentiation) computes gradients by applying the chain rule from output to input along the computation graph.

Chain Rule. For a composite function $L = L(e(a, d), c)$ where $e = a + d$ and $d = 2b$:

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e} \cdot \frac{\partial e}{\partial a} \quad (9.7)$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e} \cdot \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b} \quad (9.8)$$

Local Gradients. At each node, we compute the local gradient (derivative of output with respect to input):

For $L = c \cdot e$:

$$\frac{\partial L}{\partial e} = c, \quad \frac{\partial L}{\partial c} = e$$

For $e = a + d$:

$$\frac{\partial e}{\partial a} = 1, \quad \frac{\partial e}{\partial d} = 1$$

For $d = 2b$:

$$\frac{\partial d}{\partial b} = 2$$

Backward Pass. Starting from the output:

1. $\frac{\partial L}{\partial L} = 1$ (base case)
2. $\frac{\partial L}{\partial e} = c = -2$
3. $\frac{\partial L}{\partial c} = e = 5$
4. $\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e} \cdot \frac{\partial e}{\partial a} = (-2)(1) = -2$
5. $\frac{\partial L}{\partial d} = \frac{\partial L}{\partial e} \cdot \frac{\partial e}{\partial d} = (-2)(1) = -2$
6. $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial d} \cdot \frac{\partial d}{\partial b} = (-2)(2) = -4$

The gradients $\frac{\partial L}{\partial a} = -2$, $\frac{\partial L}{\partial b} = -4$, $\frac{\partial L}{\partial c} = 5$ can be verified analytically.

Neural Network Backpropagation. For a neural network, the same principle applies. The computation graph includes:

- Matrix multiplications ($\mathbf{W}\mathbf{x}$)
- Bias additions ($+\mathbf{b}$)
- Activation functions (σ , ReLU)
- Loss computation

The gradients of activation functions are:

Sigmoid:

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z)) \quad (9.9)$$

Tanh:

$$\frac{d \tanh(z)}{dz} = 1 - \tanh^2(z) \quad (9.10)$$

ReLU:

$$\frac{d\text{ReLU}(z)}{dz} = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases} \quad (9.11)$$

Gradient with respect to output layer. For cross-entropy loss with softmax output and correct class c :

$$\frac{\partial L}{\partial z_k} = \hat{y}_k - y_k \quad (9.12)$$

This elegant result simplifies implementation: the gradient is simply the difference between predicted and true probabilities.

9.5 Practical Considerations

Several techniques improve training:

Weight Initialization. Unlike logistic regression, neural networks cannot be initialized with all zeros (all units would compute identical gradients). Weights are initialized with small random values, typically drawn from:

$$\mathcal{N}\left(0, \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}\right)$$

where n_{in} and n_{out} are the input and output dimensions.

Regularization. To prevent overfitting:

- **L2 regularization:** Add $\lambda \|\theta\|^2$ to the loss
- **Dropout:** Randomly set unit activations to zero during training with probability p

Dropout (Hinton et al., 2012) is particularly effective. During training, each unit is "dropped" with probability p (typically 0.5). At test time, all units are used but outputs are scaled by $(1 - p)$.

Hyperparameters. Key hyperparameters include:

- Learning rate η
- Mini-batch size
- Number of hidden layers and units per layer
- Dropout rate
- Choice of optimizer (SGD, Adam, etc.)

Hyperparameters are tuned on a validation set, not the training set.

Modern Frameworks. Deep learning frameworks (PyTorch, TensorFlow) automate gradient computation via automatic differentiation. The user specifies the forward pass; gradients are computed automatically.

10 Neural Language Models

This section synthesizes embeddings and neural networks in the context of language modeling.

10.1 Feedforward Language Models

A **neural language model** uses a neural network to estimate the probability of the next word given previous words:

$$P(w_t | w_1, \dots, w_{t-1}) \tag{10.1}$$

Like n -gram models, feedforward neural LMs approximate using a fixed context window:

$$P(w_t|w_1, \dots, w_{t-1}) \approx P(w_t|w_{t-N+1}, \dots, w_{t-1}) \quad (10.2)$$

Advantages over n -gram models:

- **Generalization via embeddings:** Words with similar embeddings yield similar predictions
- **Continuous representations:** No discrete probability tables
- **Longer effective context:** Modern architectures extend beyond fixed windows

Disadvantages:

- Higher computational cost
- Requires more training data
- Less interpretable

Example. An n -gram model trained on "the cat gets fed" cannot predict "fed" after "the dog gets" without observing that exact trigram. A neural LM, knowing that "cat" and "dog" have similar embeddings, generalizes appropriately.

10.2 Forward Inference

Given context words $w_{t-N+1}, \dots, w_{t-1}$, forward inference produces a probability distribution over the next word.

Architecture. For a context window of size 3:

1. **Embedding lookup:** Each context word is represented as a one-hot vector and multiplied by the embedding matrix $\mathbf{E} \in \mathbb{R}^{d \times |V|}$:

$$\mathbf{e}_i = \mathbf{E}\mathbf{x}_i \quad (10.3)$$

2. **Concatenation:** The embeddings are concatenated to form the embedding layer:

$$\mathbf{e} = [\mathbf{e}_{t-3}; \mathbf{e}_{t-2}; \mathbf{e}_{t-1}] \quad (10.4)$$

yielding a vector of dimension $3d$.

3. **Hidden layer:** Apply weights and activation:

$$\mathbf{h} = g(\mathbf{W}\mathbf{e} + \mathbf{b}) \quad (10.5)$$

4. **Output layer:** Project to vocabulary size and apply softmax:

$$\mathbf{z} = \mathbf{U}\mathbf{h} \quad (10.6)$$

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z}) \quad (10.7)$$

The output \hat{y}_i is the predicted probability that the next word is vocabulary item i :

$$\hat{y}_i = P(w_t = V_i | w_{t-3}, w_{t-2}, w_{t-1})$$

10.3 Training Neural LMs

Training uses **self-supervision**: the next word in the corpus serves as the label.

Loss Function. For a single prediction, the loss is cross-entropy:

$$L = -\log P(w_t | w_{t-N+1}, \dots, w_{t-1}) = -\log \hat{y}_{w_t} \quad (10.8)$$

Parameter Updates. The parameters $\theta = \{\mathbf{E}, \mathbf{W}, \mathbf{b}, \mathbf{U}\}$ are updated via SGD:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \frac{\partial L}{\partial \theta} \quad (10.9)$$

Gradients are computed via backpropagation through the network, including into the embedding matrix \mathbf{E} .

Freezing vs. Fine-tuning Embeddings. Two strategies exist:

1. **Frozen embeddings:** Initialize \mathbf{E} with pre-trained embeddings (e.g., Word2Vec) and do not update during training. Only $\mathbf{W}, \mathbf{b}, \mathbf{U}$ are learned.
2. **Fine-tuning:** Initialize with pre-trained embeddings but continue updating \mathbf{E} during training. This adapts embeddings to the specific task.

Fine-tuning is generally preferred when sufficient training data is available. Freezing is useful for small datasets where updating \mathbf{E} risks overfitting.

Perplexity. Language models are evaluated using perplexity:

$$\text{PP}(W) = P(w_1, \dots, w_N)^{-1/N} = \sqrt[N]{\prod_{i=1}^N \frac{1}{P(w_i | w_1, \dots, w_{i-1})}} \quad (10.10)$$

Lower perplexity indicates a better model. Perplexity can be interpreted as the weighted average number of choices the model considers at each position.

11 Conclusion

This chapter has developed the theoretical foundations for representing word meaning and learning from text data.

Count-based methods construct word vectors from co-occurrence statistics. TF-IDF weighting and PPMI address the bias toward frequent words, yielding sparse but interpretable representations. Cosine similarity serves as the standard metric for comparing these vectors.

Word embeddings (Word2Vec, Skip-Gram with Negative Sampling) learn dense representations by training a classifier to distinguish true context words from noise samples. The resulting embeddings capture semantic and syntactic regularities, manifesting as algebraic relationships in the vector space.

Neural networks provide the computational machinery for learning embeddings and performing downstream tasks. The feedforward architecture, with its hidden layers and nonlinear activations, can represent complex functions. Training via gradient descent and backpropagation adjusts weights to minimize cross-entropy loss.

Neural language models synthesize these components, using embeddings as input to networks that predict upcoming words. These models generalize better than n -gram models because similar words yield similar predictions.

The transition from count-based to neural methods reflects a broader shift in NLP: from hand-crafted features to learned representations. The embeddings and network weights learned in this chapter form the foundation for the deep learning architectures (RNNs, Transformers) covered in subsequent chapters.

12 References

- Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, 3, 1137–1155.
- Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4), 303–314.
- Firth, J. R. (1957). A synopsis of linguistic theory 1930–1955. In *Studies in Linguistic Analysis*, 1–32. Oxford: Blackwell.
- Goldberg, Y. (2017). *Neural Network Methods for Natural Language Processing*. Morgan & Claypool Publishers.
- Harris, Z. S. (1954). Distributional structure. *Word*, 10(2–3), 146–162.
- Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*.
- Jurafsky, D., & Martin, J. H. (2024). *Speech and Language Processing* (3rd ed. draft). Chapters 6–7.
- Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. In *Proceedings of ICLR*.
- Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. In *Proceedings of ICLR Workshop*.
- Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*, 3111–3119.
- Pennington, J., Socher, R., & Manning, C. D. (2014). GloVe: Global vectors for word representation. In *Proceedings of EMNLP*, 1532–1543.
- Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533–536.
-

Appendix A: Mathematical Derivations

A.1 Derivation of Sigmoid Derivative

Given $\sigma(z) = \frac{1}{1+e^{-z}}$:

$$\frac{d\sigma}{dz} = \frac{d}{dz} (1 + e^{-z})^{-1} = -(1 + e^{-z})^{-2} \cdot (-e^{-z}) = \frac{e^{-z}}{(1 + e^{-z})^2}$$

Rewriting:

$$\frac{d\sigma}{dz} = \frac{1}{1 + e^{-z}} \cdot \frac{e^{-z}}{1 + e^{-z}} = \sigma(z) \cdot \frac{1 + e^{-z} - 1}{1 + e^{-z}} = \sigma(z)(1 - \sigma(z))$$

A.2 Gradient of Cross-Entropy with Softmax

For multi-class cross-entropy $L = -\sum_k y_k \log \hat{y}_k$ with softmax outputs $\hat{y}_k = \frac{e^{z_k}}{\sum_j e^{z_j}}$:

Taking derivative with respect to z_i :

$$\frac{\partial L}{\partial z_i} = -\sum_k y_k \frac{\partial \log \hat{y}_k}{\partial z_i} = -\sum_k y_k \frac{1}{\hat{y}_k} \frac{\partial \hat{y}_k}{\partial z_i}$$

The derivative of softmax:

$$\frac{\partial \hat{y}_k}{\partial z_i} = \begin{cases} \hat{y}_k(1 - \hat{y}_k) & k = i \\ -\hat{y}_k \hat{y}_i & k \neq i \end{cases}$$

Substituting and simplifying (where c is the correct class with $y_c = 1$):

$$\frac{\partial L}{\partial z_i} = \hat{y}_i - y_i$$

A.3 PPMI as a Matrix Factorization

Levy and Goldberg (2014) showed that Skip-Gram with Negative Sampling implicitly factorizes a shifted PMI matrix:

$$\mathbf{W}^T \mathbf{C} \approx \mathbf{M}^{\text{PMI}} - \log k$$

where k is the number of negative samples. This connection unifies count-based and prediction-based methods.

Glossary

Term	Definition
Activation Function	Nonlinear function applied element-wise to transform neural unit outputs

Term	Definition
Backpropagation	Algorithm for computing gradients by applying the chain rule backward through a computation graph
Bias	Scalar parameter added to the weighted sum in a neural unit, allowing shift of the activation threshold
Computation Graph	Directed acyclic graph representing the sequence of operations in a neural network
Cosine Similarity	Similarity measure based on the cosine of the angle between two vectors
Cross-Entropy Loss	Loss function measuring the difference between predicted probability distribution and true distribution
Distributional Hypothesis	The principle that words occurring in similar contexts have similar meanings
Dropout	Regularization technique that randomly sets unit activations to zero during training
Embedding	Dense, low-dimensional vector representation of a word
Feedforward Network	Neural network where connections flow in one direction from input to output with no cycles
Gradient Descent	Optimization algorithm that iteratively updates parameters in the direction of steepest descent
Hidden Layer	Layer between input and output that learns internal representations
IDF	Inverse Document Frequency; downweights terms appearing in many documents
Lemma	Canonical dictionary form of a word, representing a set of inflected forms
Loss Function	Function quantifying the discrepancy between predictions and ground truth
Negative Sampling	Training technique that contrasts true context words with randomly sampled "noise" words
One-Hot Vector	Sparse vector with a single 1 at the index corresponding to a word and 0s elsewhere
Perplexity	Evaluation metric for language models; the inverse probability normalized by sequence length
PMI	Pointwise Mutual Information; measures association between word pairs relative to independence
PPMI	Positive PMI; PMI clamped to non-negative values
ReLU	Rectified Linear Unit; activation function $\max(0, z)$
Self-Supervision	Training paradigm where labels are derived from the data itself (e.g., predicting the next word)
Sigmoid	S-shaped activation function mapping real values to $(0, 1)$
Skip-Gram	Word2Vec architecture that predicts context words from a target word
Softmax	Function that converts a vector of real values to a probability distribution
TF-IDF	Term Frequency-Inverse Document Frequency; weighting scheme balancing term frequency and document frequency
Word2Vec	Family of algorithms for learning word embeddings from text corpora