9.1 Word2Vec: Intuitive Understanding

Introduction

Word2Vec is a word embedding technique for Natural Language Processing published by Google in 2013. It uses a neural network model to learn word associations from large text corpora.

Key Capabilities

- Converts words into numerical vectors while preserving semantic meaning
- Detects synonym words
- Suggests additional words for partial sentences
- Identifies relationships between words (similar, opposite, related)

Word2Vec vs Traditional Methods

Traditional Methods (Bag of Words, TF-IDF)

- Generate **sparse matrices** with mostly zeros and ones
- Based on vocabulary size
- TF-IDF may include decimal values (e.g., 0.25, 0.6)

Word2Vec Approach

- Generates **dense vectors** based on feature representations
- Each word is represented by meaningful numerical values
- Captures semantic relationships between words

Feature Representation: The Core Concept

What is Feature Representation?

Each word in the vocabulary is converted into a vector based on multiple **features**. These features capture different aspects or characteristics that help define relationships between words.

Example Features

- **Gender** (masculine/feminine)

- **Royal** (royalty-related)
- **Age** (time-sensitive)
- **Food** (food-related)
- ... and many more

Dimensionality

Word2Vec typically uses **300-dimensional** feature representations, meaning each word is represented by a vector with 300 numerical values. However, this can vary (e.g., 100, 200, or 300 dimensions).

Feature Representation Example

Consider a vocabulary with words: **boy, girl, king, queen, apple, mango**
Negative values ≈ Masculine
Positive values ≈ Feminine

Each word gets numerical values for each feature based on its relationship with that feature:

Word	Gender	Royal	Age	Food	...	Feature 300
boy	-1.00	0.01	0.03	0.05	...	...
girl	+1.00	0.02	0.02	0.23	...	...
king	-0.92	0.95	0.75	0.01	...	...
queen	+0.93	0.96	0.68	0.02	...	...
apple	0.05	-0.20	0.85	0.91	...	...
mango	0.23	0.02	0.87	0.92	...	...

Understanding the Values

- **Values close to 0**: Little to no relationship (e.g., boy-royal = 0.01)
- **Opposite values**: Opposing characteristics (e.g., boy-gender = -1, girl-gender = +1)
- **Similar values**: Related words (e.g., king-royal = 0.95, queen-royal = 0.96)
- **High absolute values**: Strong relationship (e.g., apple-food = 0.91)

Vector Arithmetic: Word Relationships

One of the most powerful features of Word2Vec is the ability to perform **algebraic operations** on word vectors to discover relationships.

Famous Example

$$\text{king} - \text{boy} + \text{queen} = \text{girl}$$

This works because:

- The vector arithmetic captures the relationship "male-to-female"
- Subtracting "boy" removes the "young male" component
- Adding "queen" adds the "royal female" component
- The result is closest to "girl" (young female)

Another Example (2-Dimensional Simplification)

Let's represent words with just 2 dimensions:

$$\text{king} = [0.95, 0.96]$$

$$\text{man} = [0.95, 0.98]$$

$$\text{queen} = [0.96, 0.94]$$

Then:

$$\text{king} - \text{man} + \text{queen} \approx \text{woman}$$

Measuring Similarity: Cosine Similarity

To determine how similar two words are, we measure the **angle** between their vectors.

Cosine Similarity Formula

$$\text{Cosine Similarity} = \cos(\theta)$$

where:

- θ is the **angle** between two word vectors
- $\cos(\theta)$ gives a value between -1 and +1

Distance Formula

$\text{Distance} = 1 - \cos(\theta)$

where:

- θ is the **angle** between two word vectors
- **Distance** mathematically ranges from 0 to 2
    - When θ = 0° (identical vectors): Distance = 0

    - When θ = 90° (perpendicular): Distance = 1
    - When θ = 180° (opposite directions): Distance = 2
- **In practice**, Word2Vec distances typically fall between **0 to 1** since word vectors rarely point in exactly opposite directions
- Lower distance = more similar words
- Higher distance = less similar words

Understanding Cosine Similarity with Examples

Case 1: Similar Words (θ = 45°)

If the angle between two word vectors is **45 degrees**:

$$\cos(45^\circ) = \frac{1}{\sqrt{2}} \approx 0.7071$$

Then the distance is:

$$\text{Distance} = 1 - 0.7071 = 0.2929 \approx 0.29$$

Interpretation: Distance ≈ 0.29 indicates the words are **somewhat similar**.

Case 2: Unrelated Words (θ = 90°)

If the angle between two word vectors is **90 degrees**:

$$\cos(90^\circ) = 0$$

Then the distance is:

$$\text{Distance} = 1 - 0 = 1$$

Interpretation: Distance = 1 indicates the words are **completely different** or unrelated.

Case 3: Identical Words (θ = 0°)

If the angle between two word vectors is **0 degrees** (same direction):

$$\cos(0^\circ) = 1$$

Then the distance is:

$$\text{Distance} = 1 - 1 = 0$$

Interpretation: Distance = 0 indicates the words are **identical** or have the same meaning.

Summary of Distance Interpretation

Distance Value	Interpretation
Close to 0	Words are very similar or identical
Around 0.3	Words are somewhat similar
Around 0.5-0.7	Words have some relationship
Close to 1	Words are very different/unrelated

Real-World Applications

Recommendation Systems

Word2Vec principles apply beyond NLP. For example, in movie recommendations:

- **Feature representation** might include: genre (action, comedy), theme (comic, drama), rating
- **Movie names** are the vocabulary (e.g., "Avengers", "Iron Man")
- Movies with similar features have vectors close together
- "Avengers" and "Iron Man" would have small distance (similar movies)

Google's Pre-trained Word2Vec Model

Google released a Word2Vec model trained on:

- **3 billion words** from Google News
- **300-dimensional** vectors for each word
- Captures rich semantic relationships

Note on Feature Visibility

In large-scale trained models, individual features are **not explicitly labeled**. The neural network learns these features automatically during training, making them implicit rather than interpretable (like "gender" or "food" in our examples).

Key Takeaways

1. **Word2Vec converts words into dense numerical vectors** (typically 300 dimensions)
2. **Similar words have vectors close together** in the vector space
3. **Vector arithmetic reveals semantic relationships** (king - man + woman ≈ queen)
4. **Cosine similarity measures word similarity** by calculating the angle between vectors:

$$\text{Distance} = 1 - \cos(\theta)$$

5. **Feature representations are learned automatically** by neural networks during training
6. **Applications extend beyond NLP** to recommendation systems and other similarity-based tasks

Next Steps

To fully understand Word2Vec implementation:

- Learn about **neural networks** (Artificial Neural Networks - ANN)
- Understand **loss functions** and **optimizers**
- Study the **architecture** of Word2Vec models (CBOW and Skip-gram)
- Practice with **pre-trained models** and custom training

9.2.Word2Vec CBOW: Complete Guide

1.Introduction to Word2Vec

Word2Vec converts words into numerical vector representations using two architectures:

1. **CBOW (Continuous Bag of Words)** - Predicts target word from context
2. **Skip-gram** - Predicts context from target word

This document focuses on **CBOW**.

2.Word Embeddings Fundamentals

What Are Word Embeddings?

Dense, low-dimensional vector representations capturing semantic meaning.

Key Properties:

- Transform sparse one-hot vectors into dense meaningful vectors
- Each dimension = abstract learned feature (business-ness, formality, etc.)
- Features are **automatically discovered** during training
- Similar contexts → similar embeddings

Embedding Dimensions

Determined by **window size** (hidden layer size):

- Window size 5 → 5-dimensional vectors
- Window size 100 → 100-dimensional vectors
- Google's Word2Vec uses 300 dimensions

3.CBOW Architecture

Three-layer fully connected network:

```
Input Layer (Context words) → Hidden Layer → Output Layer (Target word)
```

Core Idea: Given context ["Ineuron", "company", "related", "to"], predict target "is"

Layer Dimensions

- **Input Layer**: Number of context words × Vocabulary size
- **Hidden Layer**: Window size (determines embedding dimension)
- **Output Layer**: Vocabulary size (probability distribution)

Weight Matrices

W_1 (Input→Hidden): Vocab size × Embedding dim

- Each row = one word's embedding vector
- **This is what we extract after training**

W_2 (Hidden→Output): Embedding dim × Vocab size

- Used for predictions during training
- **Discarded after training**

4.Training Data Preparation

Window Size Selection

- Must be **odd number** for proper center element
- Center word = **output** (target)
- Surrounding words = **input** (context)

Creating Training Pairs

Sliding window creates multiple examples:

- Input = all words except center
- Output = center word

Why bidirectional context? Model needs words before AND after target to capture semantic relationships.

5.Forward Propagation Process

Step 1: Extract Word Embeddings

Operation: $x \cdot W_1$ (one-hot vector multiplied by W_1)

Critical Understanding:

```
Dimension: (1×V) · (V×D) = (1×D)
Function: Extracts corresponding row from W₁
```

Why it works:

```
h = Σ(x[j] × W₁[j]) for all j
```

Since x is one-hot (only one element = 1):

- Only one term survives in the sum
- Effectively selects one row from W_1

Example:

```
x = [1, 0, 0, 0, 0, 0, 0]  → Selects Row 1
x = [0, 1, 0, 0, 0, 0, 0]  → Selects Row 2
```

Step 2: Average Context Embeddings

Formula: $h = (1/C) \times \Sigma(\text{embeddings of context words})$

where C = number of context words

Why average? Creates single fixed-size representation of context (Bag of Words property - order doesn't matter).

Step 3: Project to Output Space

Hidden representation multiplied by W_2 produces raw scores (logits) for each vocabulary word.

Step 4: Apply Softmax

Formula: $\text{softmax}(z_i) = e^{z_i} / \Sigma e^{z_j}$

Converts raw scores into probability distribution (all probabilities sum to 1).

6.Loss Function and Training

Cross-Entropy Loss

Formula: $L = -\log(P(\text{target}|\text{context}))$

Measures difference between:

- True target: One-hot encoded
- Predicted: Probability distribution

Goal: Minimize negative log probability of correct word.

Backpropagation

```
1. Forward pass → compute predictions
2. Calculate loss
3. Compute gradients (∂L/∂W₁, ∂L/∂W₂)
4. Update weights: W := W − α·∂L/∂W
5. Repeat thousands of times
```

Weight Evolution

Initial: Random small numbers
Training: Gradual adjustment via backpropagation
Final: W_1 contains meaningful embeddings where similar words have similar vectors

7.Complete Numerical Example

Setup

Corpus: "Ineuron company is related to data science"

Vocabulary (7 words):

```
1: Ineuron  2: company  3: is  4: related
5: to  6: data  7: science
```

Training Example:

- Window size: 5
- Context: [Ineuron, company, related, to]
- Target: is
- Embedding dimension: 5

W_1 Matrix (7×5) - Word Embeddings

```
        Dim1   Dim2   Dim3   Dim4   Dim5
Row 1: [0.92,  0.45,  0.23,  0.67,  0.34]  ← Ineuron
Row 2: [0.85,  0.12,  0.56,  0.78,  0.29]  ← company
Row 3: [0.15,  0.89,  0.34,  0.21,  0.67]  ← is
Row 4: [0.67,  0.23,  0.91,  0.45,  0.12]  ← related
Row 5: [0.34,  0.56,  0.78,  0.23,  0.45]  ← to
Row 6: [0.71,  0.33,  0.88,  0.42,  0.19]  ← data
Row 7: [0.68,  0.31,  0.85,  0.47,  0.16]  ← science
```

Row interpretation: Each row = complete 5-dimensional embedding for one word. **ROWS** provide weights for each dimension of a word's embedding:
Column interpretation: Each column = one feature across all words

Example - Row 1 (word "Ineuron"):
Each value represents a learned semantic feature:

- 0.92 → Feature 1 (e.g., business-related-ness)
- 0.45 → Feature 2 (e.g., technology-related-ness)
- 0.23 → Feature 3 (e.g., action/verb-ness)
- 0.67 → Feature 4 (e.g., formality level)
- 0.34 → Feature 5 (e.g., commonality)

Example - Column 1 (Feature 1):

```
[0.92]  ← Ineuron
[0.85]  ← company
[0.15]  ← is
[0.67]  ← related
[0.34]  ← to
[0.71]  ← data
[0.68]  ← science
```

In plain words: "This shows how strongly each word exhibits Feature 1 (e.g., business-related-ness)."

Observations:

- "Ineuron" (0.92) and "company" (0.85) score high → both are business-related
- "is" (0.15) and "to" (0.34) score low → not business-related
- Similar scores suggest similar meaning for this feature

W_2 Matrix (5×7) - Prediction Weights

```
        Ineu.  comp.  is     rela.  to     data   sci.
Row 1: [0.32,  0.45,  0.67,  0.23,  0.56,  0.12,  0.89]
Row 2: [0.21,  0.78,  0.34,  0.91,  0.45,  0.67,  0.23]
Row 3: [0.56,  0.23,  0.89,  0.12,  0.78,  0.34,  0.45]
Row 4: [0.78,  0.34,  0.12,  0.56,  0.23,  0.91,  0.67]
Row 5: [0.45,  0.91,  0.23,  0.67,  0.34,  0.78,  0.12]
```

Column interpretation: Recipe for predicting one word (contributions from all hidden dimensions)
Row interpretation: One hidden dimension's influence on all words

8.Step-by-Step Calculation

STEP 1: One-Hot Encoding

```
x_Ineuron = [1, 0, 0, 0, 0, 0, 0]
x_company = [0, 1, 0, 0, 0, 0, 0]
x_related = [0, 0, 0, 1, 0, 0, 0]
x_to      = [0, 0, 0, 0, 1, 0, 0]
```

STEP 2: Extract Embeddings ($x \cdot W_1$)

Dimension check:

```
(1×7) · (7×5) = (1×5) ✓
```

Matrix multiplication formula:

For each element i in the result (i = 1 to 5):

```
h[i] = Σ(x_Ineuron[j] × W₁[j][i]) for j = 1 to 7
```

For "Ineuron": x_Ineuron · W_1

Detailed calculation:

Element 1 (first column of W_1):

```
h[1] = (1 × 0.92) + (0 × 0.85) + (0 × 0.15) + (0 × 0.67) + (0 × 0.34) + (0 × 0.71) + (0 × 0.68)
     = 0.92 + 0 + 0 + 0 + 0 + 0 + 0
     = 0.92 ✓
```

Element 2 (second column of W_1):

```
h[2] = (1 × 0.45) + (0 × 0.12) + (0 × 0.89) + (0 × 0.23) + (0 × 0.56) + (0 × 0.33) + (0 × 0.31)
     = 0.45 + 0 + 0 + 0 + 0 + 0 + 0
     = 0.45 ✓
```

Element 3 (third column of W_1):

```
h[3] = (1 × 0.23) + (0 × 0.56) + (0 × 0.34) + (0 × 0.91) + (0 × 0.78) + (0 × 0.88) + (0 × 0.85)
     = 0.23 + 0 + 0 + 0 + 0 + 0 + 0
     = 0.23 ✓
```

Element 4 (fourth column of W_1):

```
h[4] = (1 × 0.67) + (0 × 0.78) + (0 × 0.21) + (0 × 0.45) + (0 × 0.23) + (0 × 0.42) + (0 × 0.47)
     = 0.67 + 0 + 0 + 0 + 0 + 0 + 0
     = 0.67 ✓
```

Element 5 (fifth column of W_1):

```
h[5] = (1 × 0.34) + (0 × 0.29) + (0 × 0.67) + (0 × 0.12) + (0 × 0.45) + (0 × 0.19) + (0 × 0.16)
     = 0.34 + 0 + 0 + 0 + 0 + 0 + 0
     = 0.34 ✓
```

Result: h_Ineuron = [0.92, 0.45, 0.23, 0.67, 0.34] = Row 1 of W_1 ✓

Similarly:

```
h_company = [0.85, 0.12, 0.56, 0.78, 0.29]  (Row 2)
h_related = [0.67, 0.23, 0.91, 0.45, 0.12]  (Row 4)
h_to      = [0.34, 0.56, 0.78, 0.23, 0.45]  (Row 5)
```

Step 2.1: Verification - What Did We Get?

Look at the original W_1:

```
Row 1 of W₁: [0.92, 0.45, 0.23, 0.67, 0.34] ← THIS IS WHAT WE GOT!
```

Conclusion:

> We multiplied $x \cdot W_1$ and extracted Row 1 from W_1.

Key Takeaway:

The operation $x \cdot W_1$ (where x is a one-hot vector) extracts the corresponding row from W_1. The one-hot vector acts as a row selector - it's equivalent to array indexing, but written as matrix multiplication for mathematical convenience and gradient computation.

STEP 3: Average Context Embeddings

```
h = (h_Ineuron + h_company + h_related + h_to) / 4

Dim 1: (0.92 + 0.85 + 0.67 + 0.34) / 4 = 2.78 / 4 = 0.695
Dim 2: (0.45 + 0.12 + 0.23 + 0.56) / 4 = 1.36 / 4 = 0.340
Dim 3: (0.23 + 0.56 + 0.91 + 0.78) / 4 = 2.48 / 4 = 0.620
Dim 4: (0.67 + 0.78 + 0.45 + 0.23) / 4 = 2.13 / 4 = 0.533
Dim 5: (0.34 + 0.29 + 0.12 + 0.45) / 4 = 1.20 / 4 = 0.300

h = [0.695, 0.340, 0.620, 0.533, 0.300]
```

Interpretation: Average semantic meaning of context [Ineuron, company, related, to]

STEP 4: Compute Output Scores ($h \cdot W_2$)

Dimension: $(1\times5) \cdot (5\times7) = (1\times7)$

For each word j: $z[j] = \Sigma(h[i] \times W_2[i][j])$ for i=1 to 5

W_2 = (Embedding Dimension × Vocabulary Size) = (5 × 7)

```
        Col1   Col2   Col3   Col4   Col5   Col6   Col7
        Ineu.  comp.  is     rela.  to     data   sci.  ← Words at COLUMN level
        ↓      ↓      ↓      ↓      ↓      ↓      ↓
Row 1: [0.32,  0.45,  0.67,  0.23,  0.56,  0.12,  0.89]  ← Hidden Dim 1
Row 2: [0.21,  0.78,  0.34,  0.91,  0.45,  0.67,  0.23]  ← Hidden Dim 2
Row 3: [0.56,  0.23,  0.89,  0.12,  0.78,  0.34,  0.45]  ← Hidden Dim 3
Row 4: [0.78,  0.34,  0.12,  0.56,  0.23,  0.91,  0.67]  ← Hidden Dim 4
Row 5: [0.45,  0.91,  0.23,  0.67,  0.34,  0.78,  0.12]  ← Hidden Dim 5
        ↑
(because each column produces a score for ONE word)
```

W_2 Matrix: What Each Row and Column Represents

COLUMNS (Vertical)

Each column = All the weights needed to predict ONE specific word

Example - Column 1 (word "Ineuron"):

```
[0.32]  ← How much Hidden Dim 1 contributes
[0.21]  ← How much Hidden Dim 2 contributes
[0.56]  ← How much Hidden Dim 3 contributes
[0.78]  ← How much Hidden Dim 4 contributes (strongest)
[0.45]  ← How much Hidden Dim 5 contributes
```

Calculation:

$$\text{Score(Ineuron)} = h_1\times0.32 + h_2\times0.21 + h_3\times0.56 + h_4\times0.78 + h_5\times0.45$$

In plain words: "To predict 'Ineuron', take these specific amounts from each hidden dimension and add them up."

ROWS (Horizontal)

Each row = How ONE hidden dimension influences ALL words

Example - Row 1 (Hidden Dimension 1):

```
[0.32,  0.45,  0.67,  0.23,  0.56,  0.12,  0.89]
  ↓      ↓      ↓      ↓      ↓      ↓      ↓
Ineu.  comp.   is    rela.   to    data   sci.
```

In plain words: "When Hidden Dim 1 has a high value, it pushes strongly toward 'science' (0.89) and 'is' (0.67), but weakly toward 'data' (0.12)."

If $h_1 = 0.9$, then:

- Contributes 0.9 × 0.89 = 0.801 to "science" score
- Contributes 0.9 × 0.67 = 0.603 to "is" score
- Contributes 0.9 × 0.12 = 0.108 to "data" score

Summary

View	Represents
Column	Recipe for predicting one word (contributions from all 5 hidden dims)
Row	One hidden dimension's voting power across all 7 words

Matrix multiplication $h \cdot W_2$: Takes your hidden values and uses both perspectives simultaneously to compute scores for all words at once.

The Calculation: $h \cdot W_2$

For each output neuron (each word in vocabulary), we compute:

```
z[j] = Σ(h[i] × W₂[i][j]) for i = 1 to 5
```

Output for "Ineuron" (Column 1 of W_2)

```
z₁ = (0.695 × 0.32) + (0.340 × 0.21) + (0.620 × 0.56) + (0.533 × 0.78) + (0.300 × 0.45)
   = 0.2224 + 0.0714 + 0.3472 + 0.4157 + 0.1350
   = 1.1917
```

Output for "company" (Column 2 of W_2)

```
z₂ = (0.695 × 0.45) + (0.340 × 0.78) + (0.620 × 0.23) + (0.533 × 0.34) + (0.300 × 0.91)
   = 0.3128 + 0.2652 + 0.1426 + 0.1812 + 0.2730
   = 1.1748
```

Output for "is" (Column 3 of W_2)

```
z₃ = (0.695 × 0.67) + (0.340 × 0.34) + (0.620 × 0.89) + (0.533 × 0.12) + (0.300 × 0.23)
   = 0.4657 + 0.1156 + 0.5518 + 0.0640 + 0.0690
   = 1.2661
```

Output for "related" (Column 4 of W_2)

```
z₄ = (0.695 × 0.23) + (0.340 × 0.91) + (0.620 × 0.12) + (0.533 × 0.56) + (0.300 × 0.67)
   = 0.1599 + 0.3094 + 0.0744 + 0.2985 + 0.2010
   = 1.0432
```

Output for "to" (Column 5 of W_2)

```
z₅ = (0.695 × 0.56) + (0.340 × 0.45) + (0.620 × 0.78) + (0.533 × 0.23) + (0.300 × 0.34)
   = 0.3892 + 0.1530 + 0.4836 + 0.1226 + 0.1020
   = 1.2504
```

Output for "data" (Column 6 of W_2)

```
z₆ = (0.695 × 0.12) + (0.340 × 0.67) + (0.620 × 0.34) + (0.533 × 0.91) + (0.300 × 0.78)
   = 0.0834 + 0.2278 + 0.2108 + 0.4850 + 0.2340
   = 1.2410
```

Output for "science" (Column 7 of W_2)

```
z₇ = (0.695 × 0.89) + (0.340 × 0.23) + (0.620 × 0.45) + (0.533 × 0.67) + (0.300 × 0.12)
   = 0.6186 + 0.0782 + 0.2790 + 0.3571 + 0.0360
   = 1.3689
```

Raw Output Scores (Logits)

```
z = [1.1917, 1.1748, 1.2661, 1.0432, 1.2504, 1.2410, 1.3689]
      ↑       ↑       ↑       ↑       ↑       ↑       ↑
   Ineuron company  is    related  to     data   science
```

Interpretation: These are raw scores (logits) before being converted to probabilities. Higher values indicate the model thinks that word is more likely to be the target.

STEP 5: Apply Softmax

Formula:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{7} e^{z_j}}$$

Step 5.1: Calculate e^z for each element

```
e^z₁ = e^1.1917 = 3.2927
e^z₂ = e^1.1748 = 3.2373
e^z₃ = e^1.2661 = 3.5467
e^z₄ = e^1.0432 = 2.8384
e^z₅ = e^1.2504 = 3.4918
e^z₆ = e^1.2410 = 3.4594
e^z₇ = e^1.3689 = 3.9304
```

Step 5.2: Sum all e^z values

```
Sum = 3.2927 + 3.2373 + 3.5467 + 2.8384 + 3.4918 + 3.4594 + 3.9304
    = 23.7967
```

Step 5.3: Divide each e^z by the sum

```
P(Ineuron | context) = 3.2927 / 23.7967 = 0.1384 = 13.84%
P(company | context) = 3.2373 / 23.7967 = 0.1361 = 13.61%
P(is | context)      = 3.5467 / 23.7967 = 0.1491 = 14.91%
P(related | context) = 2.8384 / 23.7967 = 0.1193 = 11.93%
P(to | context)      = 3.4918 / 23.7967 = 0.1468 = 14.68%
P(data | context)    = 3.4594 / 23.7967 = 0.1454 = 14.54%
P(science | context) = 3.9304 / 23.7967 = 0.1652 = 16.52% ← Highest!
```

Final Output Probabilities

```
ŷ = [0.1384, 0.1361, 0.1491, 0.1193, 0.1468, 0.1454, 0.1652]
      ↑       ↑       ↑       ↑       ↑       ↑       ↑
   Ineuron company  is    related  to     data   science
```

Verification: Sum = 0.1384 + 0.1361 + 0.1491 + 0.1193 + 0.1468 + 0.1454 + 0.1652 = 1.0003 ≈ 1.0 ✓

Interpretation:

- The model predicts "science" with highest probability (16.52%)
- But the true target word is "is" (14.91%)
- The model is wrong! It needs more training.

STEP 6: Compare with True Target

True target: "is" (position 3)

```
y = [0, 0, 1, 0, 0, 0, 0]
```

Predicted Probabilities

```
ŷ = [0.1384, 0.1361, 0.1491, 0.1193, 0.1468, 0.1454, 0.1652]
                      ↑                                 ↑
              True target = "is"              Model's prediction = "science"
              (14.91%)                        (16.52% - WRONG!)
```

Analysis:

- True word: "is" (position 3)
- Model assigned 14.91% probability to "is"
- Model predicted "science" with 16.52% (highest probability)
- Model is incorrect - needs more training!

STEP 7: Calculate Loss

Cross-Entropy Loss Formula:

$$\mathcal{L} = -\sum_{j=1}^{7} y_j \log(\hat{y}_j)$$

Since y is one-hot encoded (only position 3 = 1, rest = 0):

$$\mathcal{L} = -\log(\hat{y}_3) = -\log(0.1491)$$

Calculation

```
log(0.1491) = -1.9025

Loss = -(-1.9025) = 1.9025
```

Interpretation:

- Loss = 1.9025 (moderately high)
- Lower loss = better prediction
- If model predicted "is" with 100% confidence: Loss = -log(1) = 0 (perfect!)
- Current loss of 1.9025 indicates significant room for improvement

STEP 8: Backpropagation (Weight Updates)

After calculating the loss, we update the weights using gradient descent.

Gradient Descent Formula

```
W₁_new = W₁_old - (learning_rate × ∂Loss/∂W₁)
W₂_new = W₂_old - (learning_rate × ∂Loss/∂W₂)
```

How Gradients Are Calculated

For W_2:

```
∂Loss/∂W₂ᵢⱼ = (ŷⱼ − yⱼ) × hᵢ
```

Where:

- $\hat{y}_j$ = predicted probability for word j
- y_j = true label (1 if target, 0 otherwise)
- h_i = hidden layer value at position i

For W_1:
The gradient is computed via chain rule, backpropagating through W_2.

Example Weight Update

Let's update one weight in W_1:

```
Current weight: w₁₁ = 0.92 (first element of Ineuron's embedding)
Gradient: ∂Loss/∂w₁₁ = −0.05 (calculated via backpropagation)
Learning rate: α = 0.01

New weight = 0.92 − (0.01 × −0.05)
           = 0.92 + 0.0005
           = 0.9205
```

Interpretation:

- Negative gradient (-0.05) means increasing this weight will reduce loss
- So we increase it slightly by 0.0005

The Training Loop

This process repeats for:

- **All 35 weights in W_1** (7 words × 5 dimensions)
- **All 35 weights in W_2** (5 dimensions × 7 words)
- **Thousands of training examples**
- **Multiple epochs**

Eventually, weights converge to values that minimize the loss!

9.Key Theoretical Insights

1. Window Size = Embedding Dimension

Choice of window size directly determines final vector dimensionality.

2. Context-Based Learning

Fundamental assumption: **words in similar contexts have similar meanings**

3. Bag of Words Property

Order doesn't matter in CBOW - model averages context embeddings.

4. Automatic Feature Discovery

Network discovers meaningful features during training without manual specification.

5. Semantic Space Properties

After training:

- Similar words cluster together
- Relationships expressible as vector arithmetic
- Example: king - man + woman ≈ queen

6. From Random to Meaningful

Meaningful representations emerge from:

- Random initialization
- Simple objective (predict center word)
- Large unlabeled text
- Gradient descent optimization

10.What Gets Kept vs Discarded

After Training:

W_1 (KEPT):

- Contains word embeddings we want
- Each row = complete embedding for one word
- Used for all downstream NLP tasks

W_2 (DISCARDED):

- Only needed during training for predictions
- No longer useful once embeddings learned

Hidden Layer (TEMPORARY):

- Different for every training example
- Just computational workspace

Output Layer (TEMPORARY):

- Used during training to calculate loss
- Not needed after training

11.Complete Flow Summary

```
STEP 1: One-Hot Encoding
Context: [Ineuron, company, related, to]
→ [[1,0,0,0,0,0,0], [0,1,0,0,0,0,0], [0,0,0,1,0,0,0], [0,0,0,0,1,0,0]]

STEP 2: Extract Embeddings (x·W₁)
→ [[0.92,0.45,0.23,0.67,0.34], [0.85,0.12,0.56,0.78,0.29],
   [0.67,0.23,0.91,0.45,0.12], [0.34,0.56,0.78,0.23,0.45]]

STEP 3: Average Embeddings
→ [0.695, 0.340, 0.620, 0.533, 0.300]

STEP 4: Multiply by W₂
→ [1.1917, 1.1748, 1.2661, 1.0432, 1.2504, 1.2410, 1.3689]

STEP 5: Apply Softmax
→ [0.1384, 0.1361, 0.1491, 0.1193, 0.1468, 0.1454, 0.1652]

STEP 6: Compare with Target
True: [0, 0, 1, 0, 0, 0, 0] (is)
Pred: Model predicted "science" (16.52%) – WRONG!

STEP 7: Calculate Loss = 1.9025

STEP 8: Update W₁ and W₂ via gradient descent

REPEAT thousands of times...
```

12.Mathematical Formulation

Forward Pass

```
1. h_i = x_i · W₁              Extract embedding (1×V)·(V×D) = (1×D)
2. h = (1/C) Σ h_i             Average context
3. z = h · W₂                  Compute scores (1×D)·(D×V) = (1×V)
4. ŷ = softmax(z)              Probabilities
5. L = −log(ŷ_target)          Loss
```

Backpropagation

```
∂L/∂z = ŷ − y
∂L/∂W₂ = h^T · (ŷ − y)
∂L/∂h = (ŷ − y) · W₂^T
∂L/∂W₁ = (1/C) Σ x_i^T · ∂L/∂h

W₂ := W₂ − α·∂L/∂W₂
W₁ := W₁ − α·∂L/∂W₁
```

13.Common Questions

Q: Why $x \cdot W_1$ and not $x \cdot W_1$^T?

Answer: The correct operation is $x \cdot W_1$ (no transpose)

- x: (1×7) row vector
- W_1: (7×5) embedding matrix
- Result: (1×5) embedding vector
- Dimension check: (1×7)·(7×5) = (1×5) ✓

Q: What's the difference between W_1 and W_2?

W_1:

- Shape: (vocab_size × embedding_dim)
- Each row = word embedding
- **KEPT after training**

W_2:

- Shape: (embedding_dim × vocab_size)
- Maps embeddings to vocabulary

- **DISCARDED after training**

Q: Why average context embeddings?

- Creates single fixed-size representation
- Treats all context words equally
- Simple and efficient approach
- Alternative: weighted averaging (attention)

Q: How is CBOW different from Skip-gram?

CBOW:

- Input: Context words → Output: Center word
- Faster training
- Better for frequent words

Skip-gram:

- Input: Center word → Output: Context words
- Slower training
- Better for rare words and larger datasets

14.Practical Implementation Notes

Memory Efficiency

In practice, one-hot vectors aren't created:

```
# Conceptual (what we showed):
embedding = x @ W1

# Actual implementation:
embedding = W1[word_index]  # Direct indexing
```

Batch Processing

Process multiple examples simultaneously using matrix operations for GPU efficiency.

Negative Sampling

For large vocabularies, approximate full softmax by sampling only a few negative examples.

15.Dimension Reference

```
V = Vocabulary size = 7
D = Embedding dimension = 5
C = Context size = 4

W₁: (V×D) = (7×5)
W₂: (D×V) = (5×7)

x:  (1×V) = (1×7)  one-hot
h:  (1×D) = (1×5)  embedding
z:  (1×V) = (1×7)  logits
ŷ:  (1×V) = (1×7)  probabilities

x·W₁: (1×7)·(7×5) = (1×5)
h·W₂: (1×5)·(5×7) = (1×7)
```

16.Applications

Word embeddings enable:

- Similarity calculations (cosine similarity)
- NLP tasks (classification, sentiment analysis, NER)
- Semantic operations (king - man + woman ≈ queen)
- Transfer learning for downstream tasks

Pre-trained vs Training:

- **Pre-trained:** Google's Word2Vec trained on billions of words
- **Custom:** Train on domain-specific data for specialized vocabulary

Real-world applications use large corpora (millions of words) for high-quality embeddings.

9.3 Word2Vec Skip-gram: Complete Guide

1.Introduction to Skip-gram

Skip-gram is the second architecture in Word2Vec that learns word embeddings by predicting **context words from a target word** - the inverse of CBOW.

Key Difference from CBOW

Aspect	CBOW	Skip-gram
Input	Multiple context words	Single target word
Output	Single target word	Multiple context words
Prediction	Context → Center	Center → Context
Training Speed	Faster	Slower
Best For	Frequent words	Rare words, larger datasets

2.Skip-gram Architecture

Three-layer fully connected network:

```
Input Layer (Target word) → Hidden Layer → Output Layer (Context words)
```

Core Idea: Given input word "is", predict context ["Ineuron", "company", "related", "to"]

Layer Dimensions

- **Input Layer**: Vocabulary size (one-hot encoded target word)
- **Hidden Layer**: Window size (embedding dimension)
- **Output Layer**: Vocabulary size × Number of context words

Weight Matrices

W_1 (Input→Hidden): Vocab size × Embedding dim

- Each row = one word's embedding vector
- **This is what we extract after training**

W_2 (Hidden→Output): Embedding dim × Vocab size

- Used for predictions during training
- **Discarded after training**

Critical Difference: Skip-gram produces **multiple outputs** (one for each context position), while CBOW produces a single output.

3.Training Data Preparation

Window Size Selection

- Must be **odd number** for proper center element
- Center word = **input** (target)
- Surrounding words = **output** (context)

Creating Training Pairs

Corpus: "Ineuron company is related to data science"

Window size 5 example:

```
Window 1: [Ineuron, company, is, related, to]
  Input:  is (center)
  Output: Ineuron, company, related, to

Window 2: [company, is, related, to, data]
  Input:  related (center)
  Output: company, is, to, data

Window 3: [is, related, to, data, science]
  Input:  to (center)
  Output: is, related, data, science
```

Key Point: Each training example generates **multiple output predictions** (one for each context word).

4.Forward Propagation Process

Step 1: One-Hot Encode Target Word

Input: Single target word (e.g., "is")

$$\mathbf{x}_{\text{is}} = [0, 0, 1, 0, 0, 0, 0]$$

where:

- $\mathbf{x}$ = one-hot vector
- Length = vocabulary size
- Only position 3 = 1 (for "is")

Note: In Skip-gram terminology, "is" is the **input word**, and we're trying to predict the **output context words** [Ineuron, company, related, to].

Step 2: Extract Word Embedding

Operation:

$$\mathbf{h} = \mathbf{x} \cdot \mathbf{W}_1$$

where:

- $\mathbf{x}$ = $(1 \times V)$ one-hot vector
- $\mathbf{W}_1$ = $(V \times D)$ embedding matrix
- $\mathbf{h}$ = $(1 \times D)$ embedding vector

Dimension check: $(1 \times 7) \cdot (7 \times 5) = (1 \times 5)$ ✓

What happens: The one-hot multiplication extracts the corresponding row from $\mathbf{W}_1$

Example:

$$\mathbf{h} = [0, 0, 1, 0, 0, 0, 0] \cdot \mathbf{W}_1 = \text{Row 3 of } \mathbf{W}_1$$

No averaging needed - we use the embedding directly (unlike CBOW).

Step 3: Generate Multiple Outputs

Critical Difference: Skip-gram predicts **each context word independently**.

For each context position c (where $c = 1, 2, \ldots, C$):

$$\mathbf{z}^{(c)} = \mathbf{h} \cdot \mathbf{W}_2$$

where:

- $\mathbf{h}$ = $(1 \times D)$ hidden layer
- $\mathbf{W}_2$ = $(D \times V)$ output weight matrix
- $\mathbf{z}^{(c)}$ = $(1 \times V)$ raw scores for context position c

We perform this calculation C times (once for each context word).

Step 4: Apply Softmax

For each context position c:

$$\hat{\mathbf{y}}^{(c)} = \text{softmax}(\mathbf{z}^{(c)})$$

where:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{V} e^{z_j}}$$

Components:

- z_i = raw score for word i
- e^{z_i} = exponential of raw score
- $\sum_{j=1}^{V} e^{z_j}$ = sum of all exponentials (normalization)

Result: Probability distribution over vocabulary for each context position.

5.Loss Function

Multi-Output Cross-Entropy

Skip-gram has **multiple targets** (all context words), so we sum their losses:

$$\mathcal{L} = -\sum_{c=1}^{C} \log P(\text{context}_c | \text{target})$$

where:

- C = number of context words
- context_c = the c-th context word
- target = center (input) word

Expanded form:

$$\mathcal{L} = -\sum_{c=1}^{C} \sum_{j=1}^{V} y_j^{(c)} \log(\hat{y}_j^{(c)})$$

where:

- $y_j^{(c)}$ = 1 if word j is the c-th context word, 0 otherwise
- $\hat{y}_j^{(c)}$ = predicted probability for word j at position c

Simplified (since $y^{(c)}$ is one-hot):

$$\mathcal{L} = -\sum_{c=1}^{C} \log(\hat{y}_{\text{true}_c}^{(c)})$$

6.Complete Numerical Example

Setup

Corpus: "Ineuron company is related to data science"

Vocabulary (7 words):

```
1: Ineuron  2: company  3: is  4: related
5: to  6: data  7: science
```

Training Example:

- Window size: 5
- Input: **is** (center word)
- Target Context: [Ineuron, company, related, to]
- Embedding dimension: 5

W_1 Matrix (7×5) - Word Embeddings

```
        Dim1   Dim2   Dim3   Dim4   Dim5
Row 1: [0.92,  0.45,  0.23,  0.67,  0.34]  ← Ineuron
Row 2: [0.85,  0.12,  0.56,  0.78,  0.29]  ← company
Row 3: [0.15,  0.89,  0.34,  0.21,  0.67]  ← is (TARGET)
Row 4: [0.67,  0.23,  0.91,  0.45,  0.12]  ← related
Row 5: [0.34,  0.56,  0.78,  0.23,  0.45]  ← to
Row 6: [0.71,  0.33,  0.88,  0.42,  0.19]  ← data
Row 7: [0.68,  0.31,  0.85,  0.47,  0.16]  ← science
```

W_2 Matrix (5×7) - Prediction Weights

```
        Ineu.  comp.  is     rela.  to     data   sci.
Row 1: [0.32,  0.45,  0.67,  0.23,  0.56,  0.12,  0.89]
Row 2: [0.21,  0.78,  0.34,  0.91,  0.45,  0.67,  0.23]
Row 3: [0.56,  0.23,  0.89,  0.12,  0.78,  0.34,  0.45]
Row 4: [0.78,  0.34,  0.12,  0.56,  0.23,  0.91,  0.67]
Row 5: [0.45,  0.91,  0.23,  0.67,  0.34,  0.78,  0.12]
```

7.Step-by-Step Calculation

STEP 1: One-Hot Encoding

Input word: "is" (position 3)

$\mathbf{x}_{\text{is}} = [0, 0, 1, 0, 0, 0, 0]$

STEP 2: Extract Embedding

Operation: $\mathbf{h} = \mathbf{x}_{\text{is}} \cdot \mathbf{W}_1$

Dimension: $(1 \times 7) \cdot (7 \times 5) = (1 \times 5)$ ✓

Calculation:

For each element i in the result ($i = 1$ to 5):

$$h[i] = \sum_{j=1}^{7} x_{\text{is}}[j] \times W_1[j][i]$$

Since x_{is} has only position 3 = 1:

Element 1:

$$h[1] = (0 \times 0.92) + (0 \times 0.85) + (1 \times 0.15) + (0 \times 0.67) + (0 \times 0.34) + (0 \times 0.71) + (0 \times 0.68)$$
$$h[1] = 0.15$$

Element 2:

$$h[2] = (0 \times 0.45) + (0 \times 0.12) + (1 \times 0.89) + (0 \times 0.23) + (0 \times 0.56) + (0 \times 0.33) + (0 \times 0.31)$$
$$h[2] = 0.89$$

Element 3:

$$h[3] = (0 \times 0.23) + (0 \times 0.56) + (1 \times 0.34) + (0 \times 0.91) + (0 \times 0.78) + (0 \times 0.88) + (0 \times 0.85)$$
$$h[3] = 0.34$$

Element 4:

$$h[4] = (0 \times 0.67) + (0 \times 0.78) + (1 \times 0.21) + (0 \times 0.45) + (0 \times 0.23) + (0 \times 0.42) + (0 \times 0.47)$$
$$h[4] = 0.21$$

Element 5:

$$h[5] = (0 \times 0.34) + (0 \times 0.29) + (1 \times 0.67) + (0 \times 0.12) + (0 \times 0.45) + (0 \times 0.19) + (0 \times 0.16)$$

$$h[5] = 0.67$$

Result:

$$\mathbf{h} = [0.15, 0.89, 0.34, 0.21, 0.67] = \text{Row 3 of } \mathbf{W}_1$$

✓

Interpretation: This is the embedding representation of the word "is".

STEP 3: Compute Output Scores

Key Point: We compute scores for **all 4 context words independently**.

Operation: $\mathbf{z}^{(c)} = \mathbf{h} \cdot \mathbf{W}_2$

Dimension: $(1 \times 5) \cdot (5 \times 7) = (1 \times 7)$ ✓

For each word j in vocabulary:

$$z[j] = \sum_{i=1}^{5} h[i] \times W_2[i][j]$$

Output Scores for ALL Context Positions

Important: In Skip-gram, we use the **same** $\mathbf{W}_2$ matrix to predict all context words.

The calculation is:

$$z[j] = (0.15 \times W_2[1][j]) + (0.89 \times W_2[2][j]) + (0.34 \times W_2[3][j]) + (0.21 \times W_2[4][j]) + (0.67 \times W_2[5][j])$$

Score for "Ineuron" (Column 1)

$$z_1 = (0.15 \times 0.32) + (0.89 \times 0.21) + (0.34 \times 0.56) + (0.21 \times 0.78) + (0.67 \times 0.45)$$

$$z_1 = 0.0480 + 0.1869 + 0.1904 + 0.1638 + 0.3015$$

$$z_1 = 0.8906$$

Score for "company" (Column 2)

$$z_2 = (0.15 \times 0.45) + (0.89 \times 0.78) + (0.34 \times 0.23) + (0.21 \times 0.34) + (0.67 \times 0.91)$$

$$z_2 = 0.0675 + 0.6942 + 0.0782 + 0.0714 + 0.6097$$

$$z_2 = 1.5210$$

Score for "is" (Column 3)

$$z_3 = (0.15 \times 0.67) + (0.89 \times 0.34) + (0.34 \times 0.89) + (0.21 \times 0.12) + (0.67 \times 0.23)$$

$$z_3 = 0.1005 + 0.3026 + 0.3026 + 0.0252 + 0.1541$$

$$z_3 = 0.8850$$

Score for "related" (Column 4)

$$z_4 = (0.15 \times 0.23) + (0.89 \times 0.91) + (0.34 \times 0.12) + (0.21 \times 0.56) + (0.67 \times 0.67)$$

$$z_4 = 0.0345 + 0.8099 + 0.0408 + 0.1176 + 0.4489$$

$$z_4 = 1.4517$$

Score for "to" (Column 5)

$$z_5 = (0.15 \times 0.56) + (0.89 \times 0.45) + (0.34 \times 0.78) + (0.21 \times 0.23) + (0.67 \times 0.34)$$

$$z_5 = 0.0840 + 0.4005 + 0.2652 + 0.0483 + 0.2278$$

$$z_5 = 1.0258$$

Score for "data" (Column 6)

$$z_6 = (0.15 \times 0.12) + (0.89 \times 0.67) + (0.34 \times 0.34) + (0.21 \times 0.91) + (0.67 \times 0.78)$$

$$z_6 = 0.0180 + 0.5963 + 0.1156 + 0.1911 + 0.5226$$

$$z_6 = 1.4436$$

Score for "science" (Column 7)

$$z_7 = (0.15 \times 0.89) + (0.89 \times 0.23) + (0.34 \times 0.45) + (0.21 \times 0.67) + (0.67 \times 0.12)$$

$$z_7 = 0.1335 + 0.2047 + 0.1530 + 0.1407 + 0.0804$$

$$z_7 = 0.7123$$

Raw Output Scores (Logits)

$$\mathbf{z} = [0.8906, 1.5210, 0.8850, 1.4517, 1.0258, 1.4436, 0.7123]$$

```
Position:  1       2       3      4       5      6       7
Word:    Ineuron company  is   related  to    data  science
```

Interpretation: These raw scores will be converted to probabilities via softmax. We use the **same scores** for predicting all 4 context positions.

STEP 4: Apply Softmax

Formula:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{7} e^{z_j}}$$

where:

- e^{z_i} = exponential of score i
- $\sum_{j=1}^{7} e^{z_j}$ = sum of all exponentials (normalization factor)

Calculate e^{z_i} for each element

$$e^{z_1} = e^{0.8906} = 2.4364$$

$$e^{z_2} = e^{1.5210} = 4.5770$$

$$e^{z_3} = e^{0.8850} = 2.4226$$

$$e^{z_4} = e^{1.4517} = 4.2702$$

$$e^{z_5} = e^{1.0258} = 2.7895$$

$$e^{z_6} = e^{1.4436} = 4.2355$$

$$e^{z_7} = e^{0.7123} = 2.0386$$

Sum all exponentials

$$\text{Sum} = 2.4364 + 4.5770 + 2.4226 + 4.2702 + 2.7895 + 4.2355 + 2.0386$$

$$\text{Sum} = 22.7698$$

Divide each exponential by sum

$$P(\text{Ineuron}|\text{is}) = \frac{2.4364}{22.7698} = 0.1070 = 10.70\%$$

$$P(\text{company}|\text{is}) = \frac{4.5770}{22.7698} = 0.2010 = 20.10\%$$

$$P(\text{is}|\text{is}) = \frac{2.4226}{22.7698} = 0.1064 = 10.64\%$$

$$P(\text{related}|\text{is}) = \frac{4.2702}{22.7698} = 0.1875 = 18.75\%$$

$$P(\text{to}|\text{is}) = \frac{2.7895}{22.7698} = 0.1225 = 12.25\%$$

$$P(\text{data}|\text{is}) = \frac{4.2355}{22.7698} = 0.1860 = 18.60\%$$

$$P(\text{science}|\text{is}) = \frac{2.0386}{22.7698} = 0.0895 = 8.95\%$$

Final Output Probabilities

$$\hat{\mathbf{y}} = [0.1070, 0.2010, 0.1064, 0.1875, 0.1225, 0.1860, 0.0895]$$

```
Word:     Ineuron company  is   related  to    data  science
Prob:     10.70%  20.10%  10.64% 18.75% 12.25% 18.60% 8.95%
```

Verification:

$$0.1070 + 0.2010 + 0.1064 + 0.1875 + 0.1225 + 0.1860 + 0.0895 = 0.9999 \approx 1.0$$

✓

Interpretation: This probability distribution is used to predict **each of the 4 context words**.

STEP 5: Compare with True Context Words

True context words:

1. Ineuron (position 1)
2. company (position 2)
3. related (position 4)
4. to (position 5)

One-hot encodings:

$$\mathbf{y}^{(1)} = [1,0,0,0,0,0,0] \quad (\text{Ineuron})$$

$$\mathbf{y}^{(2)} = [0,1,0,0,0,0,0] \quad (\text{company})$$

$$\mathbf{y}^{(3)} = [0,0,0,1,0,0,0] \quad (\text{related})$$

$$\mathbf{y}^{(4)} = [0,0,0,0,1,0,0] \quad (\text{to})$$

STEP 6: Calculate Loss

Loss formula:

$$\mathcal{L} = -\sum_{c=1}^{4} \log P(\text{context}_c|\text{is})$$

Since we have 4 context words:

$$\mathcal{L} = -\left[\log(0.1070) + \log(0.2010) + \log(0.1875) + \log(0.1225)\right]$$

Individual Loss Terms

Loss for "Ineuron":

$$\mathcal{L}_1 = -\log(0.1070) = -(-2.2349) = 2.2349$$

Loss for "company":

$$\mathcal{L}_2 = -\log(0.2010) = -(-1.6038) = 1.6038$$

Loss for "related":

$$\mathcal{L}_3 = -\log(0.1875) = -(-1.6736) = 1.6736$$

Loss for "to":

$$\mathcal{L}_4 = -\log(0.1225) = -(-2.0994) = 2.0994$$

Total Loss

$$\mathcal{L} = 2.2349 + 1.6038 + 1.6736 + 2.0994 = 7.6117$$

Interpretation:

- Total loss = 7.6117 (quite high)
- Average loss per context word = 7.6117 / 4 = 1.9029
- Compare to CBOW loss (1.9025) - very similar!

- Model needs significant training to improve

Why is loss high?

- Model predicted "company" with highest probability (20.10%)
- But all 4 context words should have been predicted with high confidence
- Current predictions are spread across vocabulary

STEP 7: Backpropagation

Gradient Descent Formula:

$$\mathbf{W}_1^{\text{new}} = \mathbf{W}_1^{\text{old}} - \alpha \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{W}_1}$$

$$\mathbf{W}_2^{\text{new}} = \mathbf{W}_2^{\text{old}} - \alpha \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{W}_2}$$

where:

- α = learning rate (e.g., 0.01)
- $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}$ = gradient of loss with respect to weights

Gradient Calculation for W_2

For Skip-gram with multiple context words:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_2} = \sum_{c=1}^{4} \mathbf{h}^T \cdot (\hat{\mathbf{y}} - \mathbf{y}^{(c)})$$

where:

- $\mathbf{h}^T$ = (5×1) transposed hidden layer
- $(\hat{\mathbf{y}} - \mathbf{y}^{(c)})$ = (1×7) error for context word c
- Result = (5×7) gradient matrix

The gradients accumulate across all 4 context predictions.

Gradient Calculation for W_1

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_1} = \sum_{c=1}^{4} \mathbf{x}_{\text{is}}^T \cdot \left[(\hat{\mathbf{y}} - \mathbf{y}^{(c)}) \cdot \mathbf{W}_2^T \right]$$

where:

- $\mathbf{x}_{\text{is}}^T$ = (7×1) transposed one-hot input
- $[(\hat{\mathbf{y}} - \mathbf{y}^{(c)}) \cdot \mathbf{W}_2^T]$ = backpropagated error
- Result = (7×5) gradient matrix

Example Weight Update

Current weight: $w_{3,1} = 0.15$ (first element of "is" embedding)

Computed gradient: $\frac{\partial \mathcal{L}}{\partial w_{3,1}} = 0.08$

Learning rate: $\alpha = 0.01$

Update:

$$w_{3,1}^{\text{new}} = 0.15 - (0.01 \times 0.08) = 0.15 - 0.0008 = 0.1492$$

Interpretation:

- Positive gradient (0.08) means increasing this weight increases loss
- So we decrease it by a small amount (0.0008)
- After thousands of updates, weights converge to optimal values

8.Key Differences: CBOW vs Skip-gram

Architecture Comparison

Feature	CBOW	Skip-gram
Input	Multiple context words	Single target word
Hidden Layer	Average of context embeddings	Direct target embedding
Output	Single prediction	Multiple predictions
Loss	Single cross-entropy	Sum of multiple cross-entropies
Gradients	From one output	Accumulated from multiple outputs

Mathematical Comparison

CBOW Forward Pass:

$$\mathbf{h} = \frac{1}{C}\sum_{c=1}^{C}(\mathbf{x}^{(c)} \cdot \mathbf{W}_1)$$

$$\mathbf{z} = \mathbf{h} \cdot \mathbf{W}_2$$

$$\mathcal{L} = -\log P(\text{target}|\text{context})$$

Skip-gram Forward Pass:

$$\mathbf{h} = \mathbf{x}_{\text{target}} \cdot \mathbf{W}_1$$

$$\mathbf{z}^{(c)} = \mathbf{h} \cdot \mathbf{W}_2 \quad (\text{same for all } c)$$

$$\mathcal{L} = -\sum_{c=1}^{C} \log P(\text{context}_c|\text{target})$$

Training Efficiency

CBOW:

- Fewer training examples (one per window)
- Faster training
- Example count = Number of windows

Skip-gram:

- More training examples (C per window)
- Slower training
- Example count = Number of windows × C

For our corpus: With 3 windows and C=4 context words:

- CBOW: 3 training examples
- Skip-gram: 3 × 4 = 12 training examples

9.When to Use CBOW vs Skip-gram

Use CBOW When:

✅ **You have frequent words**

- CBOW smooths over context by averaging

- Works well when words appear many times
- Distributional information is reliable

✅ **You need faster training**

- One prediction per window
- Less computational cost
- Good for smaller datasets

✅ **Word order doesn't matter much**

- Averaging loses positional information
- Fine for many NLP tasks

✅ **You have limited computational resources**

Example Use Cases:

- Sentiment analysis with common vocabulary
- Document classification
- Quick prototyping and experimentation

Use Skip-gram When:

✅ **You have rare words**

- Each context position gets its own prediction
- Rare words get multiple training signals
- Better representation for infrequent words

✅ **You have larger datasets**

- More training examples from same corpus
- Can handle the computational overhead
- Better results worth the extra time

✅ **You need higher quality embeddings**

- Generally produces better embeddings
- Especially for rare words
- Industry standard for pre-trained models

✅ **Semantic relationships are critical**

- Better at capturing subtle relationships
- More training signals per word

Example Use Cases:

- Large-scale pre-trained embeddings (like Google's Word2Vec)
- Domain-specific terminology (medical, legal)
- Tasks requiring fine-grained semantic distinctions
- Rare word understanding

Performance Comparison

Metric	CBOW	Skip-gram
Training Speed	Fast	Slow
Memory Usage	Lower	Higher
Rare Word Quality	Lower	Higher
Frequent Word Quality	Good	Excellent
Semantic Relationships	Good	Excellent
Scalability	Limited	Better

Practical Recommendation

Start with:

- CBOW for initial experiments and small datasets
- Skip-gram for production systems and large corpora

Google's Word2Vec uses: Skip-gram with negative sampling on 100 billion words

10.Complete Flow Summary

```
STEP 1: One-Hot Encoding
Target: is
→ [0, 0, 1, 0, 0, 0, 0]

STEP 2: Extract Embedding (x·W₁)
→ [0.15, 0.89, 0.34, 0.21, 0.67] (Row 3 of W₁)

STEP 3: Multiply by W₂
→ [0.8906, 1.5210, 0.8850, 1.4517, 1.0258, 1.4436, 0.7123]

STEP 4: Apply Softmax
→ [0.1070, 0.2010, 0.1064, 0.1875, 0.1225, 0.1860, 0.0895]

STEP 5: Compare with 4 Context Words
True: [Ineuron, company, related, to]
Pred: Same probability distribution used for all 4

STEP 6: Calculate Loss for ALL 4 Context Words
Loss = -(log(0.1070) + log(0.2010) + log(0.1875) + log(0.1225))
     = 2.2349 + 1.6038 + 1.6736 + 2.0994
     = 7.6117

STEP 7: Backpropagate through ALL 4 outputs
Gradients accumulate from all context predictions

STEP 8: Update W₁ and W₂ via gradient descent

REPEAT for ALL windows × ALL context words...
```

11.Mathematical Formulation Summary

Forward Pass

Step 1: Extract target embedding

$$\mathbf{h} = \mathbf{x}_{\text{target}} \cdot \mathbf{W}_1$$

Dimension: $(1 \times V) \cdot (V \times D) = (1 \times D)$

Step 2: Compute scores for each context position

$$\mathbf{z}^{(c)} = \mathbf{h} \cdot \mathbf{W}_2 \quad \text{for } c = 1, 2, \ldots, C$$

Dimension: $(1 \times D) \cdot (D \times V) = (1 \times V)$

Step 3: Apply softmax

$$\hat{\mathbf{y}}^{(c)} = \text{softmax}(\mathbf{z}^{(c)})$$

where:

$$\text{softmax}(z_i)^{(c)} = \frac{e^{z_i^{(c)}}}{\sum_{j=1}^{V} e^{z_j^{(c)}}}$$

Step 4: Calculate loss

$$\mathcal{L} = -\sum_{c=1}^{C} \log P(\text{context}_c | \text{target})$$

Expanded:

$$\mathcal{L} = -\sum_{c=1}^{C} \sum_{j=1}^{V} y_j^{(c)} \log(\hat{y}_j^{(c)})$$

Simplified (one-hot encoding):

$$\mathcal{L} = -\sum_{c=1}^{C} \log(\hat{y}_{\text{true}_c}^{(c)})$$

Backpropagation

Output gradient:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(c)}} = \hat{\mathbf{y}}^{(c)} - \mathbf{y}^{(c)}$$

Gradient for $\mathbf{W}_2$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_2} = \sum_{c=1}^{C} \mathbf{h}^T \cdot (\hat{\mathbf{y}}^{(c)} - \mathbf{y}^{(c)})$$

Dimension: $(D \times 1) \cdot (1 \times V) = (D \times V)$

Gradient for hidden layer:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}} = \sum_{c=1}^{C} (\hat{\mathbf{y}}^{(c)} - \mathbf{y}^{(c)}) \cdot \mathbf{W}_2^T$$

Dimension: $(1 \times V) \cdot (V \times D) = (1 \times D)$

Gradient for $\mathbf{W}_1$:

$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_1} = \mathbf{x}_{\text{target}}^T \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{h}}$

Dimension: $(V \times 1) \cdot (1 \times D) = (V \times D)$

Weight updates:

$\mathbf{W}_1 := \mathbf{W}_1 - \alpha \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{W}_1}$

$\mathbf{W}_2 := \mathbf{W}_2 - \alpha \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{W}_2}$

where α = learning rate

12.Skip-gram Variants

Standard Skip-gram (What We Covered)

- Predicts all context words independently
- Uses full softmax over vocabulary
- Computationally expensive for large vocabularies

Hierarchical Softmax

Problem: Full softmax requires computing probabilities for all V words.

Solution: Use binary tree structure to reduce complexity.

Complexity:

- Full softmax: $O(V)$
- Hierarchical softmax: $O(\log V)$

How it works:

- Words are leaves of binary tree
- Each prediction = path from root to leaf
- Probability = product of binary decisions

Negative Sampling (Most Popular)

Problem: Computing gradients for all vocabulary words is slow.

Solution: Sample only a few "negative" examples.

Modified objective:

$$\mathcal{L} = -\log \sigma(\mathbf{v}_{\text{context}}^T \mathbf{v}_{\text{target}}) - \sum_{i=1}^{k} \mathbb{E}_{w_i \sim P_n(w)} \left[\log \sigma(-\mathbf{v}_{w_i}^T \mathbf{v}_{\text{target}})\right]$$

where:

- σ = sigmoid function
- k = number of negative samples (typically 5-20)
- $P_n(w)$ = noise distribution for sampling

Benefits:

- Much faster training
- Only updates k negative words + 1 positive
- Similar quality to full softmax

Google's Word2Vec implementation uses negative sampling.

13.Dimension Reference

```
V = Vocabulary size = 7
D = Embedding dimension = 5
C = Context size = 4 (number of context words)

W₁: (V × D) = (7 × 5)
W₂: (D × V) = (5 × 7)

x:  (1 × V) = (1 × 7)  one-hot target
h:  (1 × D) = (1 × 5)  embedding
z:  (1 × V) = (1 × 7)  logits (same for all c)
ŷ:  (1 × V) = (1 × 7)  probabilities (same for all c)

x·W₁: (1 × 7)·(7 × 5) = (1 × 5)
h·W₂: (1 × 5)·(5 × 7) = (1 × 7)
```

14.Common Questions

Q1: Why do all context positions use the same W_2?

Answer:

- Skip-gram assumes **positional independence**
- The relationship between target and context is symmetric
- Same embedding should predict all nearby words
- This is the "bag of words" assumption - position doesn't matter

Alternative: Some models use position-specific weights, but this increases parameters significantly.

Q2: How is loss accumulated across context words?

Answer: We simply **sum** the individual losses:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_1 + \mathcal{L}_2 + \mathcal{L}_3 + \mathcal{L}_4$$

Each $\mathcal{L}_c$ is the cross-entropy loss for predicting context word c.

Q3: Why is Skip-gram better for rare words?

Answer:

- **More training signals:** Rare word appears once but generates C training examples
- **CBOW averaging:** Dilutes the signal from rare context words
- **Skip-gram:** Each context word gets full attention

Example:

- Word appears 10 times in corpus
- CBOW: 10 training examples
- Skip-gram: 10 × 4 = 40 training examples

Q4: What if window includes the target word itself?

Answer:

- We **skip** the center position when creating context
- Context = all words in window **except** center
- Otherwise model would just learn identity mapping

Q5: How does gradient accumulation work?

Answer:
For each training example with C context words:

1. Compute prediction (same for all c)
2. Calculate error for each context: $\hat{\mathbf{y}} - \mathbf{y}^{(c)}$
3. Sum errors: $\sum_{c=1}^{C}(\hat{\mathbf{y}} - \mathbf{y}^{(c)})$
4. Use summed error for backpropagation

This means frequent context words have **stronger gradients**.

15.Practical Implementation Notes

Memory Efficiency

Don't create one-hot vectors explicitly:

```
# Conceptual (what we showed):
h = x @ W1

# Actual implementation:
h = W1[target_word_index]  # Direct indexing
```

Batch Processing

Process multiple target words simultaneously:

```
# Shape: (batch_size, embedding_dim)
h_batch = W1[target_indices]

# Shape: (batch_size, vocab_size)
z_batch = h_batch @ W2
```

Subsampling Frequent Words

Very frequent words (like "the", "is") provide less information. Skip-gram often **subsamples** them:

$$P(\text{discard word } w) = 1 - \sqrt{\frac{t}{f(w)}}$$

where:

- $f(w)$ = frequency of word w
- t = threshold (e.g., 10^{-5})

Effect: More training focus on meaningful words.

Dynamic Window Size

Instead of fixed window size, **randomly sample** window size for each example:

- Helps model learn different context ranges
- Words closer to target get more weight
- Improves generalization

16.Visualization of Skip-gram Training

Training Example Flow

```
Corpus: "Ineuron company is related to data science"

Window 1: [Ineuron, company, is, related, to]
├─ Target: is
└─ Generates 4 training pairs:
   ├─ (is → Ineuron)
   ├─ (is → company)
   ├─ (is → related)
   └─ (is → to)

Window 2: [company, is, related, to, data]
├─ Target: related
└─ Generates 4 training pairs:
   ├─ (related → company)
   ├─ (related → is)
   ├─ (related → to)
   └─ (related → data)

Window 3: [is, related, to, data, science]
├─ Target: to
└─ Generates 4 training pairs:
   ├─ (to → is)
   ├─ (to → related)
   ├─ (to → data)
   └─ (to → science)

Total: 12 training pairs from 3 windows
```

Weight Update Flow

```
For each training pair (target → context):

1. Extract target embedding from W₁
   ↓
2. Multiply by W₂ to get scores
   ↓
3. Apply softmax to get probabilities
   ↓
4. Calculate loss for this context word
   ↓
5. Compute gradients
   ↓
6. Accumulate gradients (if multiple contexts)
   ↓
7. Update W₁ and W₂

After processing ALL pairs:
└─ W₁ contains learned word embeddings
```

17.Embedding Quality Analysis

What Makes Good Embeddings?

Semantic Similarity:

Similar words should have small cosine distance:

$$\text{similarity}(\mathbf{v}_{\text{king}}, \mathbf{v}_{\text{queen}}) = \frac{\mathbf{v}_{\text{king}} \cdot \mathbf{v}_{\text{queen}}}{||\mathbf{v}_{\text{king}}|| \cdot ||\mathbf{v}_{\text{queen}}||}$$

Relationship Preservation:

Vector arithmetic should capture relationships:

$$\mathbf{v}_{\text{king}} - \mathbf{v}_{\text{man}} + \mathbf{v}_{\text{woman}} \approx \mathbf{v}_{\text{queen}}$$

Skip-gram Embedding Properties

After training on large corpus:

Clusters form naturally:

- Countries group together

- Verbs cluster by tense
- Adjectives by sentiment
- Technical terms by domain

Analogies work:

- Paris : France :: London : ? → England
- Walking : Walked :: Swimming : ? → Swam
- Good : Better :: Bad : ? → Worse

Distance reflects semantics:

- distance(cat, dog) < distance(cat, theory)
- distance(company, business) < distance(company, science)

18.Applications and Extensions

Direct Applications

1. Word Similarity:

```
similar_words = find_similar("king", embeddings, top_k=5)
# Returns: [queen, prince, monarch, royal, throne]
```

2. Analogy Completion:

```
analogy("man", "woman", "king", embeddings)
# Returns: queen
```

3. Semantic Clustering:

```
clusters = kmeans(embeddings, n_clusters=10)
# Groups semantically similar words
```

Downstream Tasks

1. Text Classification:

- Represent document as average of word embeddings
- Feed to classifier (SVM, neural network)

2. Named Entity Recognition:

- Use embeddings as features
- Capture semantic context

3. Machine Translation:

- Initialize encoder/decoder with pre-trained embeddings
- Transfer semantic knowledge

4. Sentiment Analysis:

- Embeddings capture positive/negative sentiment
- "good", "excellent" cluster together

Modern Extensions

1. FastText (Facebook):

- Represents words as bags of character n-grams
- Handles out-of-vocabulary words
- Better for morphologically rich languages

2. GloVe (Stanford):

- Combines Skip-gram with matrix factorization
- Uses global word co-occurrence statistics
- Often competitive with Word2Vec

3. Contextualized Embeddings:

- BERT, GPT, ELMo
- Different embeddings based on context
- "bank" (river) vs "bank" (money)

19.Training Tips and Best Practices

Hyperparameter Selection

Embedding Dimension:

- Small datasets: 50-100 dimensions
- Medium datasets: 100-300 dimensions
- Large datasets: 300-500 dimensions
- **Google's Word2Vec: 300 dimensions**

Window Size:

- Small windows (2-3): Capture syntactic relationships
- Large windows (5-10): Capture semantic/topical relationships
- **Common choice: 5**

Learning Rate:

- Start: 0.025
- Linearly decay to: 0.0001
- Adjust based on loss convergence

Negative Samples:

- Small datasets: 5-20 negative samples
- Large datasets: 2-5 negative samples
- **Google's Word2Vec: 5-10**

Training Epochs:

- Typical: 5-15 epochs
- Monitor validation loss
- Early stopping to prevent overfitting

Data Preprocessing

Essential steps:

1. **Lowercase:** Convert all text to lowercase (optional)
2. **Remove punctuation:** Unless semantically important
3. **Remove stopwords:** Optional - sometimes helpful to keep
4. **Tokenization:** Split into words
5. **Minimum frequency:** Remove rare words (e.g., frequency < 5)

Advanced:

- Subword tokenization (BPE, WordPiece)
- Phrase detection ("New York" → "New_York")
- Special token handling (numbers, URLs)

Corpus Requirements

Minimum:

- At least 10,000 sentences
- At least 100,000 words

Recommended:

- 1 million+ sentences
- 10 million+ words

Optimal:

- Google's Word2Vec: 100 billion words
- Results improve with more data

Domain-specific:

- Medical corpus for medical NLP
- Legal corpus for legal NLP
- Quality > quantity for specialized domains

20.Evaluation Metrics

Intrinsic Evaluation

1. Word Similarity Tasks:

- Human-annotated word pairs with similarity scores
- Compare cosine similarity with human judgments
- Spearman correlation coefficient

Example datasets:

- WordSim-353
- SimLex-999
- MEN dataset

2. Analogy Tasks:

- Test relationships: a:b :: c:?
- Accuracy = correct predictions / total analogies

Categories:

- Semantic: king:queen :: man:woman
- Syntactic: walking:walked :: swimming:swam

Example dataset:

- Google Analogy Test Set (19,544 questions)

Extrinsic Evaluation

Test on downstream tasks:

- Text classification accuracy
- Named entity recognition F1-score
- Sentiment analysis accuracy
- Machine translation BLEU score

Better embeddings → Better downstream performance

21.Comparison Summary: CBOW vs Skip-gram

Visual Comparison

```
CBOW Architecture:
[Context₁, Context₂, Context₃, Context₄]
        ↓
  [Average Embedding]
        ↓
    [W₂ Matrix]
        ↓
   [Single Output]
        ↓
    [Target Word]

Skip-gram Architecture:
    [Target Word]
        ↓
  [Word Embedding]
        ↓
    [W₂ Matrix]
        ↓
[Multiple Outputs (same distribution)]
        ↓
[Context₁, Context₂, Context₃, Context₄]
```

Loss Comparison

CBOW:

$\mathcal{L}_{\text{CBOW}} = -\log P(\text{target}|\text{context}_1, \ldots, \text{context}_C)$

- **Single loss value** per window
- Predicts one word from many

Skip-gram:

$\mathcal{L}_{\text{Skip-gram}} = -\sum_{c=1}^{C} \log P(\text{context}_c|\text{target})$

- **Sum of C loss values** per window
- Predicts many words from one

Training Example Count

For corpus with N windows and context size C:

Model	Training Examples	Gradient Updates
CBOW	N	N
Skip-gram	$N \times C$	$N \times C$

Our example: 3 windows, $C = 4$

- CBOW: 3 examples
- Skip-gram: 12 examples

Computational Complexity

Per Window:

Operation	CBOW	Skip-gram
Forward pass	1 softmax	C softmax operations
Backward pass	1 gradient	C accumulated gradients
Time complexity	$O(V)$	$O(C \times V)$

With negative sampling: Both become $O(k)$ where k = negative samples

22.Key Takeaways

Core Concepts

1. **Skip-gram predicts context from target** - the inverse of CBOW
2. **Multiple outputs per training example** - one prediction for each context word
3. **Same probability distribution** for all context positions - positional independence assumption
4. **Loss accumulates** across all context predictions: $\mathcal{L} = -\sum_{c=1}^{C} \log P(\text{context}_c|\text{target})$
5. **No averaging** - uses target embedding directly (unlike CBOW)

Practical Insights

6. **Better for rare words** - generates C training examples per occurrence
7. **Slower but higher quality** - more training examples, better embeddings
8. **Industry standard** - Google's Word2Vec uses Skip-gram with negative sampling
9. **Scales well** - performance improves with more data
10. **Flexible architecture** - can use hierarchical softmax or negative sampling

Mathematical Properties

11. **Gradients accumulate** from all context predictions during backpropagation
12. **Same $\mathbf{W}_2$ matrix** used for all context positions
13. **Vector arithmetic works** after training on sufficient data: $\mathbf{v}_{\text{king}} - \mathbf{v}_{\text{man}} + \mathbf{v}_{\text{woman}} \approx \mathbf{v}_{\text{queen}}$
14. **Cosine similarity** measures semantic relatedness between word vectors
15. **Embeddings are dense** - every dimension contributes to meaning

23.Next Steps

To Implement Skip-gram:

1. **Understand neural networks** - forward/backward propagation
2. **Learn optimization** - gradient descent, Adam optimizer
3. **Study efficiency techniques** - negative sampling, hierarchical softmax
4. **Practice with frameworks** - Gensim, TensorFlow, PyTorch

5. **Experiment with hyperparameters** - window size, dimensions, learning rate

Advanced Topics:

- **Subword embeddings** (FastText)
- **Global matrix factorization** (GloVe)
- **Contextualized embeddings** (BERT, ELMo)
- **Multilingual embeddings** (MUSE, LASER)
- **Domain adaptation** techniques

Resources:

Original Papers:

- Mikolov et al. (2013): "Efficient Estimation of Word Representations in Vector Space"
- Mikolov et al. (2013): "Distributed Representations of Words and Phrases"

Implementations:

- Gensim library (Python)
- Word2Vec by Google (C++)
- TensorFlow/PyTorch tutorials

Pre-trained Models:

- Google News Word2Vec (3 billion words, 300 dimensions)
- FastText pre-trained vectors (157 languages)

24. Final Comparison Table

Aspect	CBOW	Skip-gram
Input	Multiple context words	Single target word
Output	Single target word	Multiple context words
Hidden Layer	Average of context embeddings	Direct target embedding
Training Examples	N (per corpus)	N × C (per corpus)
Training Speed	Fast ⚡	Slower 🐢
Rare Word Quality	Lower 📉	Higher 📈

Aspect	CBOW	Skip-gram
Frequent Word Quality	Good ✓	Excellent ✓✓
Memory Usage	Lower	Higher
Semantic Quality	Good	Excellent
Best Dataset Size	Small to Medium	Medium to Large
Computational Cost	$O(V)$ per window	$O(C \times V)$ per window
Gradient Updates	One per window	C per window
Pre-trained Models	Less common	More common ⭐
Industry Adoption	Moderate	High ⭐⭐

Remember: Both CBOW and Skip-gram learn the same $\mathbf{W}_1$ matrix containing word embeddings - they just use different training objectives to get there!