# Guidebook to Exercise 6

Prof. Dr. Ivan Habernal
Yassine Thlija

2025-11-25

# 1 Introduction to Word Embeddings

Before diving into Word2Vec, let's understand the fundamental problem: how can we represent words numerically for computers to understand? Traditional approaches like one-hot encoding create sparse, high-dimensional vectors that don't capture any semantic relationships. Word2Vec solves this by learning dense, low-dimensional vectors that preserve semantic and syntactic relationships between words.

# 2 Why Word2Vec Works

## 2.1 The Distributional Hypothesis

Word2Vec is built upon the **distributional hypothesis**, which states that "words that appear in similar contexts tend to have similar meanings." Let's break this down with examples:

- Consider the words "king" and "queen." They often appear in similar contexts:

  ```
  "The _____ ruled the kingdom."
  "The _____ sat on the throne."
  "The _____ wore a crown."
  ```

  Because both words can fill these blanks, Word2Vec learns that they should have similar vector representations.

- Similarly, "car" and "vehicle" appear in overlapping contexts:

  ```
  "I drove my _____ to work."
  "The _____ needs fuel."
  "This _____ has four wheels."
  ```

## 2.2 How Word2Vec Learns Meaning

Word2Vec trains a shallow neural network not for the primary purpose of prediction, but to learn word vectors as a byproduct. Think of it as a clever trick: we set up a fake prediction task to force the network to learn meaningful word representations.

**Training Process:**

1. Start with random vectors for each word
2. For each word-context pair in your training data, adjust vectors to make similar words closer
3. Repeat this until vectors stabilize

**Example:** When training on "The cat sat on the mat":

- The model sees that "cat" is often near "sat," "on," "the," "mat"
- It gradually moves "cat"s vector closer to these context words
- Similarly, "dog" will be near "barked," "ran," "play"

# 3 Skip-Gram Architecture

## 3.1 Intuition Behind Skip-Gram

Skip-Gram predicts context words from a **target word**. The reasoning is straightforward: if I tell you a word, you should be able to guess what words might appear around it.

**Example:** Given the target word "programming," what words might appear nearby?

```
Possible contexts: "code", "computer", "language", "software", "developer"
```

Skip-Gram is particularly effective for rare words because each occurrence provides multiple learning opportunities (predicting multiple context words).

## 3.2 Network Architecture

- **Input Layer:** A one-hot vector representing the target word. If your vocabulary has 10,000 words, this is a 10,000-dimensional vector with a single 1 and 9,999 zeros.
- **Hidden Layer:** This is where the magic happens! The hidden layer contains the embedding matrix $W$ of size $V \times D$, where $V$ is vocabulary size and $D$ is embedding dimension. When you multiply the one-hot input by this matrix, you simply select the corresponding row, which is exactly the word vector!

**The cool math behind this:** If input vector is $x$ (one-hot for word $i$), then hidden layer $h = W^T x = W_i$ (the $i$-th row of $W$, which is word $i$'s vector)

- **Output Layer:** Another matrix $W'$ of size $D \times V$ that converts the hidden layer back to vocabulary space, followed by softmax to get probabilities for each context word.

# 4 CBOW Architecture: another approach?

## 4.1 Intuition Behind CBOW

Continuous Bag of Words (CBOW) predicts a target word from its surrounding context words. This mirrors how humans often guess missing words: given the context, what word fits here?

**Example:** Given the context "The ___ sat on the mat," what word might fill the blank?

```
Possible targets: "cat", "dog", "baby", "person"
```

CBOW's averaging of context vectors acts as noise reduction, making it more stable but potentially losing subtle distinctions (sometimes missing rare words).

## 4.2 CBOW Architecture Details

- **Input Layer:** Multiple one-hot vectors for each context word around the target
- **Hidden Layer:** Computes the average of all context word vectors

$$h = \frac{1}{C} \sum_{c=1}^{C} W^T x_c$$

where $C$ is the number of context words

- **Output Layer:** Similar to Skip-Gram, predicts the target word from the averaged context representation

### 4.3 When to Use CBOW vs. Skip-Gram (take this with a grain of salt!)

| **Factor** | **Skip-Gram** | **CBOW** |
|---|---|---|
| Training Speed | Slower | Faster |
| Rare Words | Better | Worse |
| Frequent Words | Good | Excellent |
| Small Datasets | Better | Good |
| Large Datasets | Excellent | Good |

Table 1: Comparison of Skip-Gram vs. CBOW

## 5 Tokenization with spaCy

### 5.1 Why Tokenization Matters

Tokenization is the process of splitting raw text into meaningful units called tokens. Poor tokenization can ruin your Word2Vec model because the embeddings rely on consistent context extraction.

**Common Tokenization Issues:**

- "don't" → ["do", "n't"] vs. ["don't"]
- "U.S.A." → ["U.S.A."] vs. ["U", ".", "S", ".", "A", "."]
- "ice-cream" → ["ice-cream"] vs. ["ice", "cream"]

### 5.2 spaCy Tokenization Example

```
import spacy

# we load the english mode
nlp = spacy.load("en_core_web_sm")

# we tokenize the text
text = "Json studies deep learning to understand neural networks.
        He's exploring Word2Vec embeddings!"
doc = nlp(text)

# we get our actual token text (token != token.text, don't forget that)
tokens = [token.text.lower() for token in doc]

print(tokens)
```

# 6 Perplexity: Measuring Model Quality

## 6.1 What is Perplexity?

Perplexity measures how well a probability model predicts a sample. In the context of Word2Vec and language models, it tells us how "surprised" the model is by the actual data.

**Mathematical Definition:**

$$\text{Perplexity} = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log P(w_i|\text{context}_i)\right)$$

Where:

- $N$ is the number of words in test data
- $P(w_i|\text{context}_i)$ is the probability the model assigns to the actual word given its context

## 6.2 Intuitive Understanding

Think of perplexity as "the weighted average branching factor" - how many equally likely options the model thinks it has at each step:

- **Perplexity = 1**: Perfect prediction (model is always 100% certain and correct)
- **Perplexity = V**: Worst case (model thinks all V vocabulary words are equally likely)
- **Lower perplexity** = Better model

**Example:** If perplexity = 20, it means the model is as confused as if it had to choose randomly between 20 equally likely options at each step.