VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

MATHS FOUNDATION for COMPUTER SCIENCE (CO5097)

Assignment

Word Embedding

(word2vec)

Instructors *Nguyen An Khuong, CSE-HCMUT*
Tran Tuan Anh, CSE-HCMUT
Le Hong Trang, CSE-HCMUT

Group's members *Ly Minh Trung - 2570349*
Ngo Le Khoa - 2570432
Bui Minh Hieu - 2570406
Tran Duy Quang - 2410709
Nguyen Anh Tai - 2212980

Ho Chi Minh, December 2025

I. Introduction

1. Problem Statement

In traditional computer science, computers process text as strings of characters or ASCII/Unicode codes, but they do not really "understand" the meaning of words. Older methods like One-hot Encoding represent each word as a sparse vector with size equal to the dictionary size *V*. Example:

- man: [1, 0, 0, …, 0]
- woman: [0, 1, 0, …, 0]

Problem:

- Curse of Dimensionality: With large vocabularies (millions of words), vectors are too large and waste memory.
- Lack of Semantic Meaning: The dot product of any two one-hot vectors is zero (they are orthogonal). The computer cannot tell if "man" and "woman" or "king" and "queen" are semantically related.

Word Embedding Solution: The goal is to map words into a dense vector space of lower dimension (e.g., 100-300 dimensions), so that words with similar meanings are located close together in that space. Word2Vec is one of the most widely used techniques for this purpose.

2. Applications

Word Embedding is the foundation of most modern NLP systems:

- Machine Translation: Understand the similarities between vocabularies of different languages.
- Sentiment Analysis: Note that both "great" and "excellent" convey positive connotations.
- Information Retrieval: Find related documents even if the keywords are not 100% exact matches but have the same meaning.
- Analogies: Solve problems like "King - Man + Woman = ?". The resulting vector is often an approximation of the vector of "Queen".

Problems solved
This method solves the problem of representing unstructured data (text) into a structured form (numerical vectors) that Deep Learning models can optimize through Gradient Descent. It transforms the linguistic problem into a spatial geometry problem.

II. Model Architectures: Skip-gram and CBOW

1. Skip-gram Model

The Skip-gram Architecture

The Skip-gram architecture operates on a premise that is the inverse of the Continuous Bag of Words (CBOW) model. Instead of using surrounding context words to predict the center word, Skip-gram uses the **center word** to predict the surrounding **context words**. This model assumes that if we know the current word, we can generate the likely words that appear in its vicinity.

For instance, utilizing the sequence from our dataset "king is a man", if we designate "is" as the center word (w_c) with a context window size of 1, the Skip-gram model aims to maximize the probability of generating the context words "king" (w_{t-1}) and "a" (w_{t+1}) given the center word "is".

Mathematically, unlike CBOW which computes one conditional probability, Skip-gram computes the probability for each context word independently (assuming conditional independence):

$$P("king","a"|"is") = P("king"|"is") \cdot P("a"|"is")$$

Formal Definition

Similarly to CBOW, for every unique word indexed i in the vocabulary V, we define two distinct vector representations: - $v_i \in R^d$: The vector representation when the word serves as a **center** (input) word. - $u_i \in R^d$: The vector representation when the word serves as a **context** (target) word.

Suppose we have a center word w_c and we want to predict a context word w_o. The conditional probability is modeled using the Softmax function applied to the dot product of the context word vector and the center word vector:

$$P(w_o|w_c) = \frac{exp(u_o^\top v_c)}{\sum_{j=1}^{|V|} exp(u_j^\top v_c)}$$

Objective Function

Given a corpus sequence of length T, where the word at position t is denoted as $w^{(t)}$. With a context window size of m, the Skip-gram model assumes that context words are conditionally independent given the center word e.g., $P("king",\ "a"|"is") = P("king"|"is") \cdot P("a"|"is")$.

Therefore, the goal is to maximize the joint probability (**Likelihood**) of observing all context words given their center words across the entire text:

$$\prod_{t=1}^{T} \prod_{-m \le j \le m, j \ne 0} P\left(w^{(t+j)} | w^{(t)}\right)$$

However, computing the product of many probabilities often leads to numerical underflow. To solve this, we usually work with the logarithm of the likelihood. Maximizing the likelihood is equivalent to maximizing the average **Log-Likelihood**:

$$J = \frac{1}{T} \sum_{t=1}^{T} \sum_{-m \le j \le m, j \ne 0} logP\left(w^{(t+j)} | w^{(t)}\right)$$

In machine learning optimization, we typically minimize a loss function. Thus, the problem is formulated as minimizing the **Negative Log-Likelihood (Loss Function)**:

$$Loss = - \sum_{t=1}^{T} \sum_{-m \le j \le m, j \ne 0} logP\left(w^{(t+j)} | w^{(t)}\right)$$

Gradient Derivation

The training uses Stochastic Gradient Descent (SGD). For a specific center-context pair $\left(w_c, w_o\right)$, the gradient of the loss with respect to the center word vector v_c is derived as:

$$\frac{\partial Loss}{\partial v_c} = \sum_{j \in V} \left(P\left(w_j | w_c\right) - t_j\right) u_j$$

Where $t_j = 1$ if w_j is the actual context word, and 0 otherwise. This update rule essentially moves the center word vector v_c closer to the observed context word vectors u_o and away from other words in the vocabulary.

2. The Continuous Bag of Words (CBOW) Model

The Continuous Bag of Words (CBOW) architecture operates on an inverse premise compared to the Skip-gram model. Instead of using a single word to predict the context, CBOW assumes that the center (target) word is generated conditionally based on the aggregation of its surrounding context words within the text sequence.

For instance, utilizing the sequence from our dataset **"king is a man"**, if we designate **"is"** as the center word with a context window size of 1, the CBOW model computes the conditional probability of generating the center word **"is"** given its immediate context words **"king"** and **"a"** (as illustrated in **Fig. 1**). This is mathematically expressed as:

P(“is” | “king”, “a”)

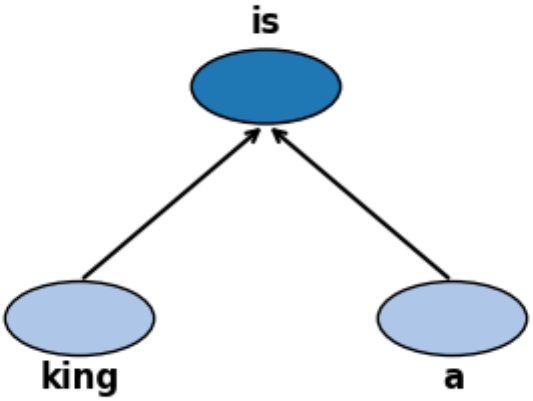


Fig. 1: CBOW model predicting “is” from context “king” and “a”

Unlike the Skip-gram model, which predicts context based on a single word, the Continuous Bag of Words (CBOW) model considers multiple context words simultaneously. To handle this, the model aggregates the information by calculating the average of the vectors corresponding to the context words.

Formally, for every unique word indexed i in the vocabulary V, we define two distinct vector representations:

- $v_i \in R^d$: The vector representation when the word serves as a **context** word.
- $u_i \in R^d$: The vector representation when the word serves as a **center** (target) word.

Suppose we want to predict a center word w_c given a surrounding context window of $2m$ words: $w_{o1}, \ldots, w_{o2m}$ (with indices $o_1, \ldots, o_{2m}$). The conditional probability P is modeled using the Softmax function applied to the dot product of the center word vector and the average vector of the context words:

$$P(w_c \mid w_{o_1}, \ldots, w_{o_{2m}}) = \frac{\exp\left(\frac{1}{2m}\mathbf{u}_c^\top(\mathbf{v}_{o_1} + \cdots + \mathbf{v}_{o_{2m}})\right)}{\sum_{i \in V} \exp\left(\frac{1}{2m}\mathbf{u}_i^\top(\mathbf{v}_{o_1} + \cdots + \mathbf{v}_{o_{2m}})\right)}$$

To simplify the notation, let $W_o = \{w_{o1}, \ldots, w_{o2m}\}$ represent the set of context words, and let $\bar{v}_o$ be their mean vector:

$$\bar{\mathbf{v}}_o = \frac{1}{2m}(\mathbf{v}_{o_1} + \cdots + \mathbf{v}_{o_{2m}})$$

The equation can then be written more concisely as:

$$P(w_c \mid \mathcal{W}_o) = \frac{\exp(\mathbf{u}_c^\top \bar{\mathbf{v}}_o)}{\sum_{i \in V} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o)}$$

Given a corpus sequence of length T, where the word at position t is denoted as $w^{(t)}$. With a context window size of m, the goal of the CBOW model is to maximize the likelihood of generating the correct center words given their respective contexts across the entire text. This likelihood function is defined as the product of probabilities at each time step:

$$\prod_{t=1}^{T} P(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)})$$

Training:

The training procedure for the CBOW model parallels that of the Skip-gram architecture. To estimate the model parameters, we employ Maximum Likelihood Estimation (MLE). Maximizing the likelihood across the entire corpus is mathematically equivalent to minimizing the global **Negative Log-Likelihood (Loss Function)**:

$$J = -\sum_{t=1}^{T} \log P(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)})$$

Expanding the logarithmic term for a specific target word w_c given its context W_o, we obtain:

$$\log P(w_c \mid \mathcal{W}_o) = \mathbf{u}_c^\top \bar{\mathbf{v}}_o - \log \left(\sum_{i \in V} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o) \right)$$

Gradient Derivation

To optimize the weights using Stochastic Gradient Descent (SGD), we need to compute the gradient of the loss function. Through differentiation with respect to any specific context word vector v_{oi} (where $i = 1, \ldots, 2m$), we arrive at the following update rule:

$$\frac{\partial \log P(w_c \mid \mathcal{W}_o)}{\partial \mathbf{v}_{o_i}} = \frac{1}{2m} \left(\mathbf{u}_c - \sum_{j \in V} \frac{\exp(\mathbf{u}_j^\top \bar{\mathbf{v}}_o)}{\sum_{i \in V} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o)} \mathbf{u}_j \right) = \frac{1}{2m} \left(\mathbf{u}_c - \sum_{j \in V} P(w_j \mid \mathcal{W}_o) \mathbf{u}_j \right)$$

Gradients for other parameters are computed similarly. An important distinction in implementation is the choice of the final embedding. Unlike the Skip-gram model, the CBOW model typically selects the context word vectors (v) to serve as the final learned word representations for downstream tasks.

III. Detailed Computations

To demonstrate the internal mechanics of the Word2Vec model, we will perform a step-by-step manual computation for the **Continuous Bag of Words (CBOW)** architecture.

1. Setup and Preprocessing

A. The Corpus We utilize the following simple corpus for demonstration:

corpus = ["king is a man", "queen is a woman", "prince is a boy", "princess is a girl", "king and queen", "prince and princess", "man and woman", "boy and girl"]

B. Vocabulary Construction

First, we tokenize the corpus and build a vocabulary V. We assign a unique index to each word.

- Vocabulary Size: $|V| = 11$
- Token-to-Index Map: {'king': 0, 'is': 1, 'a': 2, 'man': 3, 'queen': 4, 'woman': 5, 'prince': 6, 'boy': 7, 'princess': 8, 'girl': 9, 'and': 10}

C. Hyperparameters

- Embedding Dimension (d)**:** 2 (We choose 2 dimensions for easy visualization).
- Context Window (m): 1 (We look at 1 word to the left and 1 word to the right).

2. Step-by-step CBOW

Scenario: We take the sentence: "king is a man".

- Target Word (w_t)**:** "is" (Index: 1).
- Context Words (w_{ctx}) **:** "king" (Index: 0) and "a" (Index: 2).

Step 1: Initialization & Matrix Lookup

We have two weight matrices initialized with random values (for the sake of this example):

1. Input Matrix (W_{in}): Dimensions 11 x 2. Stores vectors for context words.

Hypothetical Initial Values**:** Let us assume the random initialization gave us the following vectors for our specific words:

Index	**Word**	**Char 1 (column 0)**	**Char 2 (column 1)**	**Remark**
0	**king**	**0.2**	**0.9**	this is v_{king}
1	is	-0.5	0.1	
2	**a**	**0.1**	**0.3**	this is v_a
3	man	0.8	-0.2	
...	...	...	...	

Table 1: Random

2. Output Matrix (W_{out}): Dimensions 2 x 11. Stores vectors for target words.

	king (0)	**is (1)**	**a (2)**	**man (3)**	**...**
Char 1	0.01	**0.5**	0.1	**0.1**	...
Char 2	0.99	**1.0**	0.8	**0.4**	...
Vector	u_{king}	u_{is}	u_a	u_{man}	

Step 2: Hidden Layer Projection

In CBOW, we average the input vectors of the context words to form the hidden layer vector *h*.

$$h = \frac{1}{2m}(vking + va)$$

$$h = \frac{1}{2}([0.20,\ 0.90] + [0.10,\ 0.30])$$

$$h = \frac{1}{2}[0.30,\ 1.20]$$

$$h = [0.15, 0.60]$$

The vector *h = [0.15, 0.60]* represents the aggregate meaning of the context.

Step 3: Score Calculation (Logits)

We calculate the raw scores *(z)* by taking the dot product of the hidden vector *h* with the output vectors of all words in the vocabulary.
Score of the target word “is”:

$$z_{is} = u^{\top}_{is} \cdot h = (0.50 \times 0.15) + (1.00 \times 0.60)$$

$$z_{is} = 0.075 + 0.60 = 0.675$$

Score of the non-target word “man”:

$$z_{man} = u^{\top}_{man} \cdot h = (0.10 \times 0.15) + (0.40 \times 0.60)$$

$$z_{man} = 0.015 + 0.24 = 0.255$$

With: $u^{\top}_{word}$: Transpose of vector u
h: hidden layer vector in step 2

(Note: In a full implementation, this is calculated for all 11 words in V).

Step 4: Probability Estimation (Softmax)

We apply the Softmax function to convert raw scores into probabilities. **Assuming for this simplified example that our vocabulary only consists of these two words ("is", "man"), but in real case the vocabulary must contain all words in the input corpus**:

- Exponentiate the scores:

$$e^{z_{is}} = e^{0.675} \approx 1.964$$

$$e^{z_{man}} = e^{0.255} \approx 1.290$$

- Calculate the sum:

$$\sum = 1.964 + 1.290 = 3.254$$

- Calculate Probabilities:

$$P(\text{“is”} \mid \text{context}) = \frac{1.964}{3.254} \approx 0.6035\ (60.35\%)$$

$$P(\text{“man”} \mid \text{context}) = \frac{1.290}{3.254} \approx 0.3965\ (39.65\%)$$

Step 5: Loss Calculation

We use the Cross-Entropy Loss (or Negative Log-Likelihood). Since the true label is "is" (where *y=1*), the loss depends only on the probability of the correct class.

$$\text{Loss} = -\log(P(\text{“is”} \mid \text{context}))$$

$$\text{Loss} = -\log(0.6035) \approx 0.505$$

Interpretation: The loss of 0.505 indicates the error. The training process (Backpropagation) will compute the gradients of this loss with respect to the weights in matrices W_{in} *and* W_{out} to adjust the vectors [0.20, 0.90], [0.50, 1.00], etc., making the probability of "is" closer to 1 in the next epoch.

Step 6: Backpropagation and Weight Update

After calculating the Loss, the core of the learning process involves updating the word vectors to minimize this error. We will calculate the gradients and update the vector for the context word **"king"**.

- Calculate Error Signal (e): First, we determine the error for each word in the output vocabulary. The error is the difference between the predicted probability ($\widehat{y}$) and the actual target (y).
  - Target “is” (y = 1): $e_{is} = 0.6035 - 1 = -0.3965$
  - Non-target “man” (y=0) $e_{man} = 0.3965 - 0 = 0.3965$
- Calculate Gradient w.r.t Hidden Layer (h): We backtrack the error through the Output Matrix (W_{out}). The gradient for h is the weighted sum of errors multiplied by their respective target vectors.

$$\frac{\partial Loss}{\partial h} = (e_{is} \cdot u_{is}) + (e_{man} \cdot u_{man})$$

  - Contribution from "is": -0.3965 x [0.50, 1.00] = [-0.198, -0.396]
  - Contribution from "man": 0.3965 x [0.10, 0.40] = [0.040, 0.159]
  - **Total Gradient:** [-0.198 + 0.040, -0.396 + 0.159] = [-0.158, -0.237]
- Calculate Gradient w.r.t Input Vector (v_{king}): The gradient passes through the averaging operation. Since we used 2 context words (2m=2), the gradient is distributed equally:

$$\frac{\partial Loss}{\partial vking} = \frac{\partial Loss}{\partial h} \times \frac{1}{2}$$

$$\text{Grad.}v_{king} = \frac{1}{2} \times [-0.158, -0.237] = [-0.079, -0.118]$$

- Update the Vector using Learning Rate (η): We apply the Gradient Descent update rule. Let's assume a learning rate η=0.1.

$$v_{king}^{(new)} = v_{king}^{(old)} - \eta \cdot Grad.v_{king}$$

- With $v_{king}^{(old)} = [0.20, 0.90]$

$$v_{king}^{(new)} = [0.20, 0.90] - 0.1 \times [-0.079, -0.118] = [-0.0079, -0.0118]$$

$$= [0.20, 0.90] - [-0.0079, -0.0118]$$

$$= [0.20 + 0.0079, 0.90 + 0.0118]$$

$$\approx [0.208, 0.912]$$

Conclusion of Computation: After just one training step, the vector for "king" has shifted from [0.20, 0.90] to [0.208, 0.912]. This slight adjustment effectively moves "king" into a position in the vector space that makes it better at predicting the word "is" in future iterations. Over millions of such updates, the vectors capture meaningful semantic relationships. In a full training cycle, both the context vectors (v) and target vectors (u) are adjusted to maximize their dot product.

3. Step-by-step Skip-gram

Prerequisites: We utilize the same **Corpus**, **Vocabulary** ($|V| = 11$), and **Hyperparameters** (Embedding Dimension d=2, Window Size m=1) as defined in the CBOW computation section above. We proceed directly to the manual calculation iteration.

- **Scenario:** We take the sentence: **"king is a man"**.
- **Center Word (w_c):** "is" (Index: 1).
- **Target Context Words:** "king" (Index: 0) and "a" (Index: 2). - *Note: In Skip-gram, we treat each (Center, Context) pair as a separate training sample. We will demonstrate the update for the pair **("is", "king")**.*

Step 1: Initialization & Matrix Lookup

We use two weight matrices. Note the roles: - **Input Matrix (W_{in}):** Stores vectors for words when they are **Center** words (v). - **Output Matrix (W_{out}):** Stores vectors for words when they are **Target/Context** words (u).

Hypothetical Initial Values (Consistent with previous CBOW example):

Input Matrix (W_{in}) - Center Vectors (v):

Index	Word	Char 1	Char 2	Remark
1	is	-0.5	0.1	this is v_{is}
…	…	…	…	…

Output Matrix (W_{out}) - Context Vectors (u):

Index	Word	Char 1	Char 2	Remark
0	king	0.01	0.99	this is u_{king} (Target)
3	man	0.1	0.4	this is u_{man} (Non-target)
…	…	…	…	…

(Note: We use "man" as a non-target sample to demonstrate the softmax calculation logic).

Step 2: Hidden Layer Projection

In Skip-gram, the hidden layer is simply the look-up of the center word vector. No averaging is performed.

$$h = v_{is} = [-0.5, 0.1]$$

Step 3: Score Calculation (Logits)

We calculate the raw scores (z) by taking the dot product of the center vector v_{is} (which is h) with the output vectors (u) of the words in the vocabulary.

- **Reference Formula:** $z_o = u_o^{\top} v_c$

Score for Target Word "king":

$$z_{king} = u_{king}^{\top} \cdot h = (0.01 \times -0.5) + (0.99 \times 0.1)$$

$$z_{king} = -0.005 + 0.099 = 0.094$$

Score for Non-Target Word "man":

$$z_{man} = u_{man}^{\top} \cdot h = (0.1 \times -0.5) + (0.4 \times 0.1)$$

$$z_{man} = -0.05 + 0.04 = -0.01$$

(In a full implementation, this is calculated for all 11 words in V).

Step 4: Probability Estimation (Softmax)

We apply the Softmax function to convert raw scores into probabilities. For this simplified demonstration, we assume our vocabulary subset is just {"king", "man"}.

- **Reference Formula:** $P(w_o|w_c) = \frac{exp(z_o)}{\sum_j exp(z_j)}$

Exponentiate the scores:

$$e^{z_{king}} = e^{0.094} \approx 1.0986$$

$$e^{z_{man}} = e^{-0.01} \approx 0.9900$$

Calculate the sum:

$$\sum = 1.0986 + 0.9900 = 2.0886$$

Calculate Probabilities:

$$P("king"|"is") = \frac{1.0986}{2.0886} \approx 0.5260 \quad (52.6\%)$$

$$P("man"|"is") = \frac{0.9900}{2.0886} \approx 0.4740 \quad (47.4\%)$$

Step 5: Loss Calculation

We calculate the Negative Log-Likelihood for the specific target context word "king".

- **Reference Formula:** $Loss = -\ logP(w_{target}|w_{center})$

$$Loss = -\ log(P("king"|"is"))$$

$$Loss = -\ log(0.5260) \approx 0.642$$

Interpretation: The model predicts "king" with 52.6% probability. We want this to be 100%. The loss of 0.642 represents the penalty for this uncertainty.

Step 6: Backpropagation and Weight Update

We need to update the center vector v_{is} based on the error.

Calculate Error Signal (e): Difference between predicted probability (y) and actual target (t).

$$\text{Target "king" } (t = 1): e_{king} = 0.5260 - 1 = -0.4740$$

$$\text{Non-target “man”} (t = 0): e_{man} = 0.4740 - 0 = 0.4740$$

Calculate Gradient with respect to Center Vector (v_{is}): The gradient is the weighted sum of errors and context vectors.

- **Reference Formula:** $\frac{\partial Loss}{\partial v_c} = \sum_j \left(P_j - t_j\right) u_j$
- Contribution from “king”: $-0.4740 \times [0.01, 0.99] = [-0.0047, -0.4693]$
- Contribution from “man”: $0.4740 \times [0.1, 0.4] = [0.0474, 0.1896]$

Total Gradient:

$$Grad_{v_{is}} = [-0.0047 + 0.0474, \; -0.4693 + 0.1896]$$

$$Grad_{v_{is}} = [0.0427, -0.2797]$$

Update the Vector (v_{is}): We use Stochastic Gradient Descent (SGD) with a learning rate $\eta = 0.1$.

- **Reference Formula:** $v_c^{(new)} = v_c^{(old)} - \eta \cdot \frac{\partial Loss}{\partial v_c}$

$$v_{is}(new) = [-0.5, 0.1] - 0.1 \times [0.0427, -0.2797]$$

$$v_{is}(new) = [-0.5, 0.1] - [0.00427, -0.02797]$$

$$v_{is}(new) = [-0.5 - 0.00427, 0.1 + 0.02797]$$

$$v_{is}(new) = [-0.504, 0.128]$$

Conclusion of Computation: The vector for “is” has shifted from [-0.5, 0.1] to [-0.504, 0.128]. The second component increased (from 0.1 to 0.128). Notice that the target “king” had a very high second component (0.99). The update has successfully pulled the vector of “is” closer to the direction of “king” (specifically in the second dimension) to increase the dot product in the next iteration.

IV. Corresponding Implementation in Python

1. Implementation in 2D

```
import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt
import numpy as np


# --- 1. DATASET ---
corpus = [
    "king is a man",
    "queen is a woman",
    "prince is a boy",
    "princess is a girl",
    "king and queen",
    "prince and princess",
    "man and woman",
    "boy and girl"
]
# Tạo từ điển (Create the vocabulary)
tokens = [sent.split() for sent in corpus]
vocabulary = set([word for sent in tokens for word in sent])
word2idx = {w: i for i, w in enumerate(vocabulary)}
idx2word = {i: w for w, i in word2idx.items()}
V = len(vocabulary)

print(f"Vocab size: {V}")
print(f"Word mapping: {word2idx}")

# Hyperparameters
d = 2       # Embedding dimension (2 chiều)
lr = 0.1    # Learning rate
epochs = 2000 # Train lâu một chút để tách cụm rõ (Train a little longer to
separate clusters clearly)

# Tạo training data (Create the training data)
window = 1
# Data Skip-Gram: [Target] -> [Context]
skipgram_data = []
for sent in tokens:
    for i, target in enumerate(sent):
        for j in range(max(0, i-window), min(len(sent), i+window+1)):
            if i != j:
                skipgram_data.append((word2idx[target], word2idx[sent[j]]))

# Data CBOW: [Contexts] -> [Target]
cbow_data = []
for sent in tokens:
    for i, target in enumerate(sent):
        context_idxs = []
        for j in range(max(0, i-window), min(len(sent), i+window+1)):
            if i != j:
```

```
                context_idxs.append(word2idx[sent[j]])
        if context_idxs:
            cbow_data.append((context_idxs, word2idx[target]))
```

→ Output:

Vocab size: 11

Word mapping: {'woman': 0, 'king': 1, 'a': 2, 'man': 3, 'queen': 4, 'is': 5, 'prince': 6, 'boy': 7, 'and': 8, 'princess': 9, 'girl': 10}

A. Training Skip-gram

```
print("\n--- TRAINING SKIP-GRAM (MANUAL) ---")

# 1. Khởi tạo trọng số (Random) - Initialize weights
# require_grad=True để PyTorch tự tính đạo hàm giúp (không cần viết công thức đạo hàm tay dài dòng) - Let PyTorch calculate the derivative for you (no need to write long derivative formulas)
W1 = torch.randn(V, d, requires_grad=True) # Center Vectors
W2 = torch.randn(d, V, requires_grad=True) # Context Vectors

loss_history_sg = []

for epoch in range(epochs):
    total_loss = 0

    for target_idx, context_idx in skipgram_data:
        # --- FORWARD PASS (TÍNH TAY/MANUAL) ---

        # Bước 1: Lấy vector của từ target (Lookup Table)
        # Tương đương nhân one-hot với W1: h = W1[target_idx]
        h = W1[target_idx] # Shape: [d]

        # Bước 2: Tính Score (Dot Product với toàn bộ từ trong vocab)
        # z = W2_transposed * h (nhưng ở đây W2 shape [d, V] nên nhân trực tiếp)
        # z = h . W2 -> Shape: [V]
        z = torch.matmul(h, W2)

        # Bước 3: Softmax thủ công
        # y_hat = exp(z) / sum(exp(z))
        y_hat = F.softmax(z, dim=0)

        # --- LOSS CALCULATION ---
        # Cross Entropy Loss = -log(y_hat[context_true])
        loss = -torch.log(y_hat[context_idx])
        total_loss += loss.item()

        # --- BACKWARD PASS (GRADIENT) ---
        loss.backward() # Tự động tính gradient dL/dW1 và dL/dW2
```

```
        # --- UPDATE WEIGHTS (SGD THỦ CÔNG) ---
        with torch.no_grad():
            # W = W - lr * gradient
            W1 -= lr * W1.grad
            W2 -= lr * W2.grad

            # Reset gradient về 0 cho vòng lặp sau
            W1.grad.zero_()
            W2.grad.zero_()

    if (epoch+1) % 500 == 0:
        print(f"Epoch {epoch+1}: Loss = {total_loss:.4f}")

# Lưu lại embedding kết quả để vẽ
sg_embeddings = W1.detach().numpy()
```

→ **Output:**

--- TRAINING SKIP-GRAM (MANUAL) ---

Epoch 500: Loss = 58.7819

Epoch 1000: Loss = 58.8423

Epoch 1500: Loss = 58.8509

Epoch 2000: Loss = 58.8531

B. Training CBOW

```
print("\n--- TRAINING CBOW (MANUAL) ---")

# 1. Khởi tạo lại trọng số mới
W1_cbow = torch.randn(V, d, requires_grad=True) # Context Matrix (Input)
W2_cbow = torch.randn(d, V, requires_grad=True) # Center Matrix (Output)

for epoch in range(epochs):
    total_loss = 0

    for context_idxs, target_idx in cbow_data:
        # --- FORWARD PASS (TÍNH TAY) ---

        # Bước 1: Lấy các vector context
        # contexts_vecs shape: [num_context, d]
        context_vecs = W1_cbow[context_idxs]

        # Bước 2: TÍNH TRUNG BÌNH (Đặc trưng của CBOW)
        # h = mean(context_vecs)
        h = torch.mean(context_vecs, dim=0) # Shape: [d]

        # Bước 3: Tính Score (Dot product với output matrix)
        z = torch.matmul(h, W2_cbow) # Shape: [V]
```

```
            # Bước 4: Softmax
            y_hat = F.softmax(z, dim=0)

            # --- LOSS ---
            loss = -torch.log(y_hat[target_idx])
            total_loss += loss.item()

            # --- BACKWARD ---
            loss.backward()

            # --- UPDATE ---
            with torch.no_grad():
                W1_cbow -= lr * W1_cbow.grad
                W2_cbow -= lr * W2_cbow.grad

                W1_cbow.grad.zero_()
                W2_cbow.grad.zero_()

        if (epoch+1) % 500 == 0:
            print(f"Epoch {epoch+1}: Loss = {total_loss:.4f}")

    # Lưu kết quả
    cbow_embeddings = W1_cbow.detach().numpy()
```

→ Output:

--- TRAINING CBOW (MANUAL) ---

Epoch 500: Loss = 30.4132

Epoch 1000: Loss = 30.1920

Epoch 1500: Loss = 30.0735

Epoch 2000: Loss = 29.9960

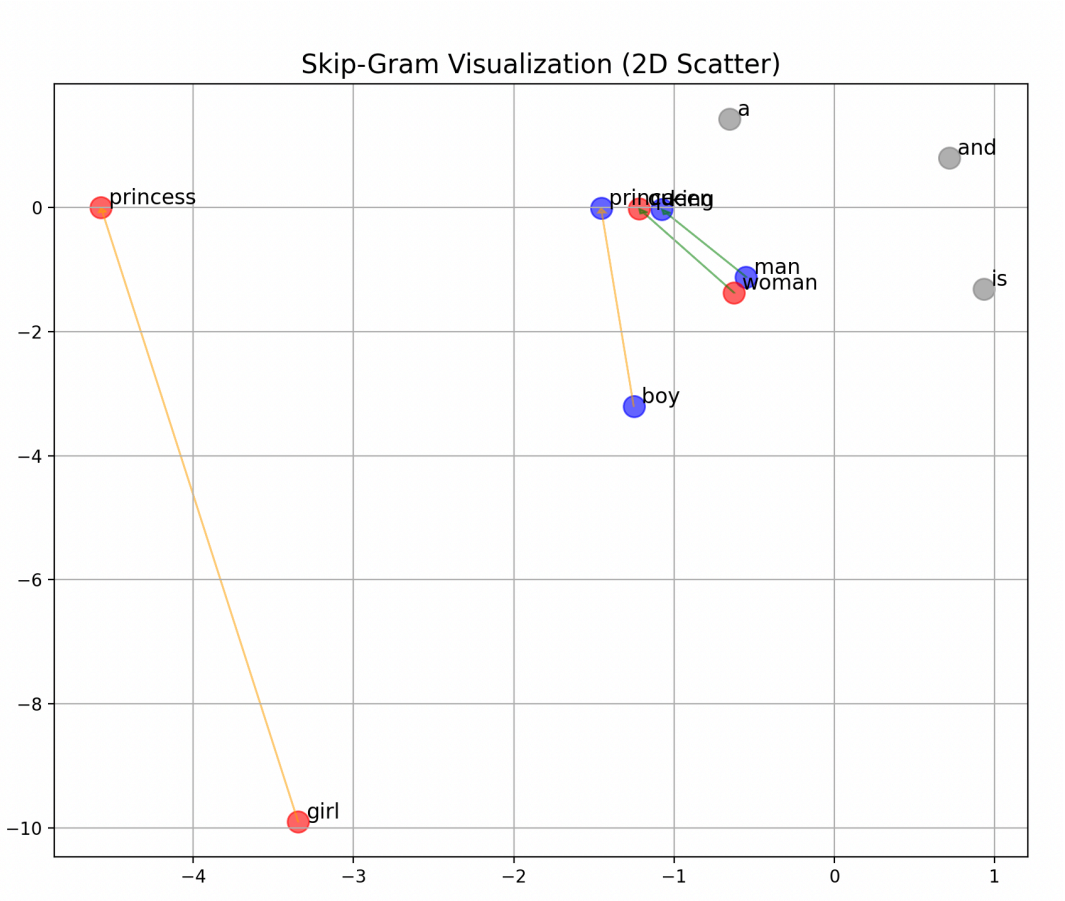
C. Visualization

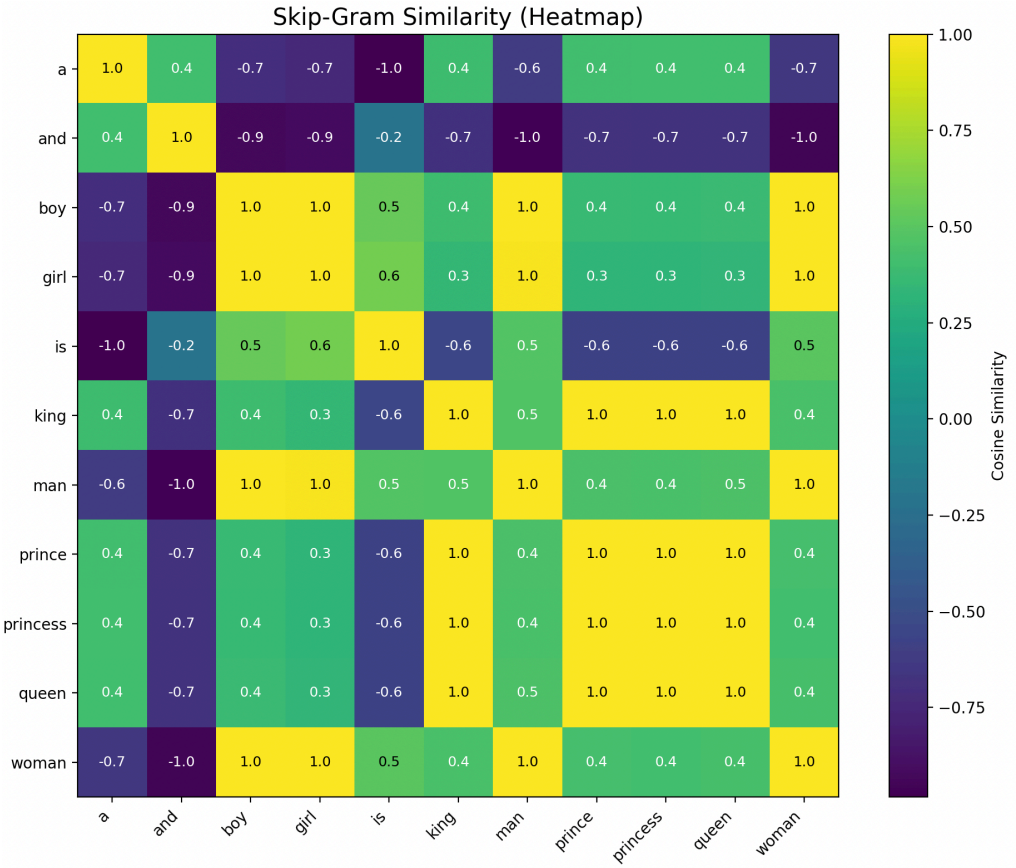
```
def plot_manual_embeddings(weights, title):
    plt.figure(figsize=(10, 8))
    plt.title(title, fontsize=15)
    plt.grid(True)

    # Vẽ các điểm
    for i in range(V):
        x, y = weights[i][0], weights[i][1]
        word = idx2word[i]

        # Tô màu khác nhau cho giới tính để dễ nhìn
        color = 'blue'
```

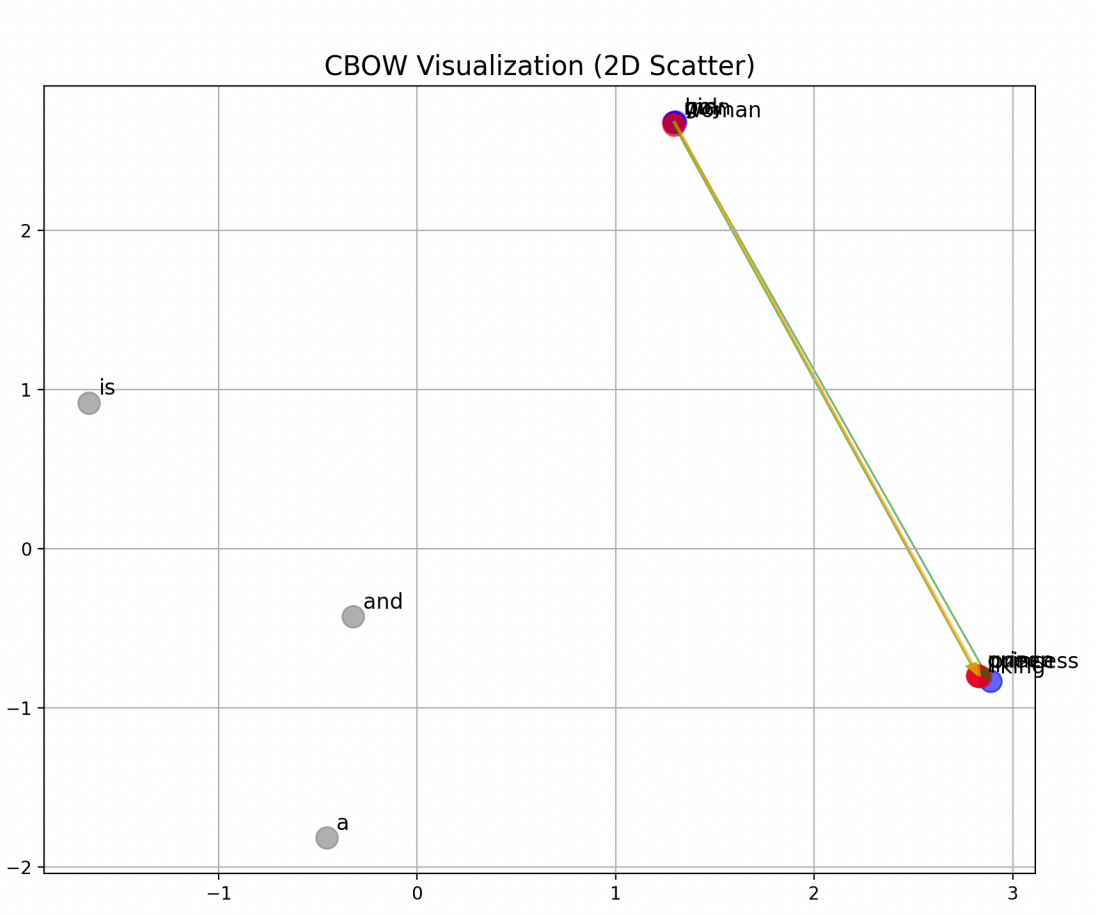
```
            if word in ['queen', 'woman', 'girl', 'princess']:
                color = 'red'
            elif word in ['king', 'man', 'boy', 'prince']:
                color = 'blue'
            else:
                color = 'gray' # từ nối

            plt.scatter(x, y, c=color, s=150, alpha=0.6)
            plt.text(x+0.05, y+0.05, word, fontsize=12)

        # Vẽ mũi tên quan hệ: Man -> King, Woman -> Queen
        # Nếu mô hình học tốt, 2 mũi tên này sẽ song song và cùng chiều dài
        def draw_arrow(w1, w2, c='green'):
            if w1 in word2idx and w2 in word2idx:
                i1, i2 = word2idx[w1], word2idx[w2]
                plt.arrow(weights[i1][0], weights[i1][1],
    weights[i2][0]-weights[i1][0],weights[i2][1]-weights[i1][1],
                          head_width=0.05, length_includes_head=True, color=c,
    alpha=0.5)

        draw_arrow("man", "king")
        draw_arrow("woman", "queen")

        # Thêm cặp khác: Boy -> Prince, Girl -> Princess
        draw_arrow("boy", "prince", c='orange')
        draw_arrow("girl", "princess", c='orange')

        plt.show()


def plot_heatmap(weights, title):
    """Vẽ Heatmap (Cosine Similarity Matrix)"""
    # 1. Tính toán Cosine Similarity Matrix
    # Normalize các vector về độ dài 1 (Unit vector)
    norms = np.linalg.norm(weights, axis=1, keepdims=True)
    # Tránh chia cho 0
    norms[norms == 0] = 1e-9
    normalized_weights = weights / norms

    # Cosine Similarity = Dot product của các unit vector
    sim_matrix = np.dot(normalized_weights, normalized_weights.T)

    # 2. Vẽ Heatmap
    plt.figure(figsize=(10, 8))
    plt.imshow(sim_matrix, cmap='viridis', interpolation='nearest')
    plt.colorbar(label='Cosine Similarity')
    plt.title(title, fontsize=15)
```

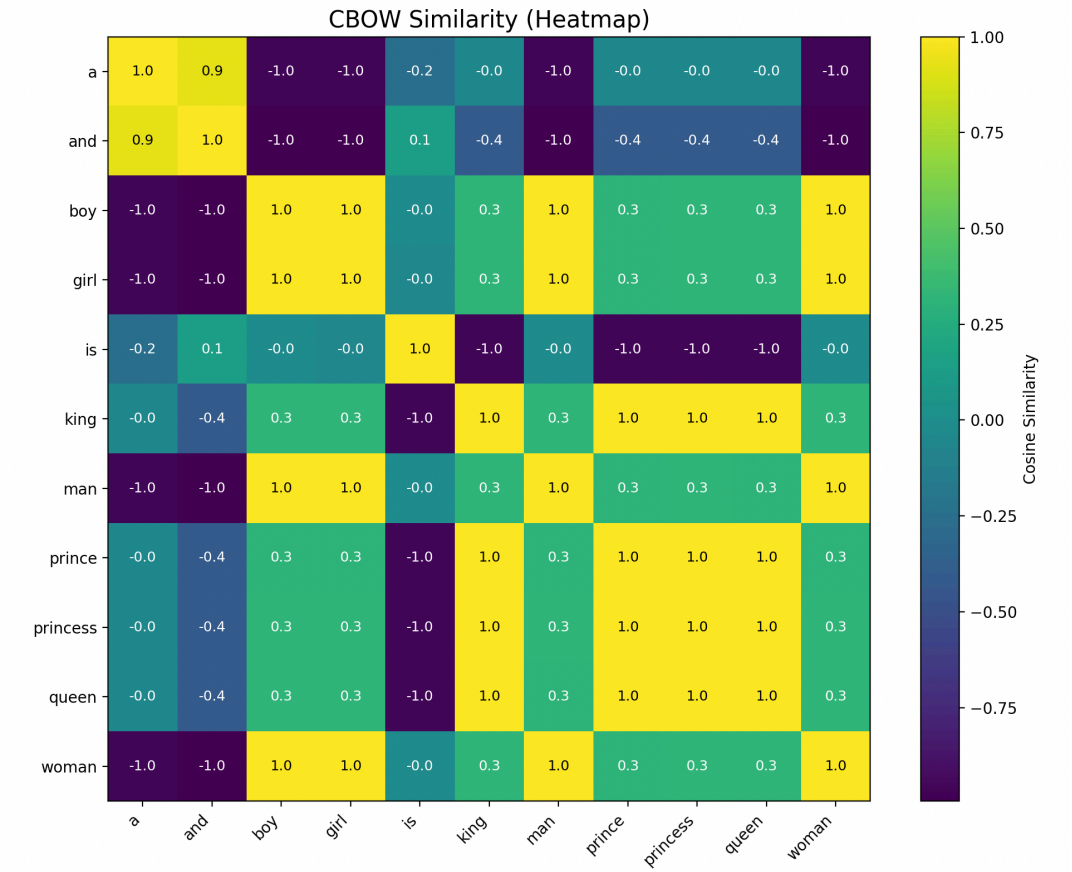
```
    # Gán nhãn trục
    tick_marks = np.arange(V)
    words = [idx2word[i] for i in range(V)]
    plt.xticks(tick_marks, words, rotation=45, ha='right')
    plt.yticks(tick_marks, words)

    # Hiển thị giá trị số lên ô
    for i in range(V):
        for j in range(V):
            val = sim_matrix[i, j]
            color = "black" if val > 0.5 else "white"
            plt.text(j, i, f"{val:.1f}", ha="center", va="center", color=color,
fontsize=9)

    plt.tight_layout()
    plt.show()

    # Vẽ đồ thị
    plot_manual_embeddings(sg_embeddings, "Skip-Gram Visualization (2D
    Scatter)")
    plot_heatmap(sg_embeddings, "Skip-Gram Similarity (Heatmap)")

    plot_manual_embeddings(cbow_embeddings, "CBOW Visualization (2D Scatter)")
    plot_heatmap(sg_embeddings, "CBOW Similarity (Heatmap)")
```

→ Output:


Skip-Gram Visualization (2D Scatter)
a
and
princess
man
woman
is
boy
girl
0
−2
−4
−6
−8
−10
−4
−3
−2
−1
0
1


Skip-Gram Similarity (Heatmap)
a 1.0 0.4 -0.7 -0.7 -1.0 0.4 -0.6 0.4 0.4 0.4 -0.7
and 0.4 1.0 -0.9 -0.9 -0.2 -0.7 -1.0 -0.7 -0.7 -0.7 -1.0
boy -0.7 -0.9 1.0 1.0 0.5 0.4 1.0 0.4 0.4 0.4 1.0
girl -0.7 -0.9 1.0 1.0 0.6 0.3 1.0 0.3 0.3 0.3 1.0
is -1.0 -0.2 0.5 0.6 1.0 -0.6 0.5 -0.6 -0.6 -0.6 0.5
king 0.4 -0.7 0.4 0.3 -0.6 1.0 0.5 1.0 1.0 1.0 0.4
man -0.6 -1.0 1.0 1.0 0.5 0.5 1.0 0.4 0.4 0.5 1.0
prince 0.4 -0.7 0.4 0.3 -0.6 1.0 0.4 1.0 1.0 1.0 0.4
princess 0.4 -0.7 0.4 0.3 -0.6 1.0 0.4 1.0 1.0 1.0 0.4
queen 0.4 -0.7 0.4 0.3 -0.6 1.0 0.5 1.0 1.0 1.0 0.4
woman -0.7 -1.0 1.0 1.0 0.5 0.4 1.0 0.4 0.4 0.4 1.0
a and boy girl is king man prince princess queen woman
Cosine Similarity
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75


CBOW Visualization (2D Scatter)
is
and
a
2
1
0
−1
−2
−1
0
1
2
3


CBOW Similarity (Heatmap)
a 1.0 0.9 -1.0 -1.0 -0.2 -0.0 -1.0 -0.0 -0.0 -0.0 -1.0
and 0.9 1.0 -1.0 -1.0 0.1 -0.4 -1.0 -0.4 -0.4 -0.4 -1.0
boy -1.0 -1.0 1.0 1.0 -0.0 0.3 1.0 0.3 0.3 0.3 1.0
girl -1.0 -1.0 1.0 1.0 -0.0 0.3 1.0 0.3 0.3 0.3 1.0
is -0.2 0.1 -0.0 -0.0 1.0 -1.0 -0.0 -1.0 -1.0 -1.0 -0.0
king -0.0 -0.4 0.3 0.3 -1.0 1.0 0.3 1.0 1.0 1.0 0.3
man -1.0 -1.0 1.0 1.0 -0.0 0.3 1.0 0.3 0.3 0.3 1.0
prince -0.0 -0.4 0.3 0.3 -1.0 1.0 0.3 1.0 1.0 1.0 0.3
princess -0.0 -0.4 0.3 0.3 -1.0 1.0 0.3 1.0 1.0 1.0 0.3
queen -0.0 -0.4 0.3 0.3 -1.0 1.0 0.3 1.0 1.0 1.0 0.3
woman -1.0 -1.0 1.0 1.0 -0.0 0.3 1.0 0.3 0.3 0.3 1.0
a and boy girl is king man prince princess queen woman
Cosine Similarity
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75

2. UseCase in invoice

A. Skip-gram

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

# --- DỮ LIỆU GIẢ LẬP (Hóa đơn mua hàng) ---
transactions = [
    ['iPhone', 'Ốp_lưng', 'Sạc_nhanh', 'AirPods'],
    ['Samsung', 'Ốp_lưng', 'Sạc_nhanh', 'GalaxyBuds'],
    ['Laptop_Dell', 'Chuột', 'Bàn_phím', 'Lót_chuột'],
    ['Macbook', 'Chuột', 'Hub_USB', 'Túi_chống_sốc'],
    ['iPhone', 'Sạc_nhanh', 'Cường_lực'],
    ['Samsung', 'Sạc_nhanh', 'Cường_lực'],
    ['Laptop_Dell', 'Chuột', 'Balo'],
    ['Bàn_phím', 'Chuột', 'Lót_chuột'],
    ['iPhone', 'AirPods', 'Apple_Watch']
]

# 1. Tạo từ điển (Vocabulary)
products = sorted(list(set([item for sublist in transactions for item in
sublist])))
prod2idx = {p: i for i, p in enumerate(products)}
idx2prod = {i: p for p, i in prod2idx.items()}
vocab_size = len(products)

print(f"Danh sách sản phẩm ({vocab_size} món): {products}")

# 2. Tạo Training Data thủ công
# Chúng ta sẽ làm Skip-Gram: Input (1 món) -> Dự đoán các món còn lại
trong cùng đơn
# Window size = 2 (Lấy 2 món bên trái, 2 món bên phải)
window_size = 2
training_data = []

for bill in transactions:
    indices = [prod2idx[p] for p in bill]
    for i, target_idx in enumerate(indices):
        # Lấy các món xung quanh trong window
        start = max(0, i - window_size)
        end = min(len(indices), i + window_size + 1)

        for j in range(start, end):
            if i != j: # Không lấy chính nó
                context_idx = indices[j]
                training_data.append((target_idx, context_idx))

print(f"\nSố lượng cặp training: {len(training_data)}")
print(f"Ví dụ 5 cặp đầu tiên (Target -> Context): {training_data[:5]}")
# Ví dụ: Mua iPhone (index X) -> Có khả năng mua Ốp lưng (index Y)
```

→ Output:

Danh sách sản phẩm (16 món): ['AirPods', 'Apple_Watch', 'Balo', 'Bàn_phím', 'Chuột', 'Cường_lực', 'GalaxyBuds', 'Hub_USB', 'Laptop_Dell', 'Lót_chuột', 'Macbook', 'Samsung', 'Sạc_nhanh', 'Túi_chống_sốc', 'iPhone', 'Ốp_lưng']

Số lượng cặp training: 70

Ví dụ 5 cặp đầu tiên (Target -> Context): [(14, 15), (14, 12), (15, 14), (15, 12), (15, 0)]

```
# --- CẤU HÌNH ---
embedding_dim = 4  # Mỗi sản phẩm được biểu diễn bằng vector 4 chiều
learning_rate = 0.05
epochs = 1000

# --- MODEL SKIP-GRAM THỦ CÔNG ---
class ManualSkipGram(nn.Module):
    def __init__(self, vocab_size, emb_dim):
        super(ManualSkipGram, self).__init__()
        # Input -> Hidden: Đây chính là ma trận vector của sản phẩm
        # bias=False để đúng với công thức toán học W.x
        self.W1 = nn.Linear(vocab_size, emb_dim, bias=False)

        # Hidden -> Output: Ma trận ngữ cảnh
        self.W2 = nn.Linear(emb_dim, vocab_size, bias=False)

    def forward(self, target_idx):
        # 1. Tạo one-hot vector cho đầu vào thủ công
        x_one_hot = torch.zeros(vocab_size)
        x_one_hot[target_idx] = 1.0

        # 2. Tính toán lớp ẩn (Lấy hàng tương ứng của W1)
        # h = W1.T * x
        hidden = self.W1(x_one_hot)

        # 3. Tính toán đầu ra (Score cho tất cả sản phẩm context)
        # z = W2 * h
        score = self.W2(hidden)

        # 4. Không cần Softmax ở đây vì CrossEntropyLoss của PyTorch đã
bao gồm Softmax
        return score

# Khởi tạo model
model = ManualSkipGram(vocab_size, embedding_dim)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

# --- TRAINING LOOP ---
print("\n--- BẮT ĐẦU TRAINING ---")
for epoch in range(epochs):
```

```
    total_loss = 0
    for target, context in training_data:

        # 1. Zero grad
        optimizer.zero_grad()

        # 2. Forward
        pred_scores = model(target) # Output: vector độ dài [vocab_size]

        # 3. Loss (So sánh vector dự đoán với index thực tế của context)
        loss = criterion(pred_scores.unsqueeze(0), torch.tensor([context]))

        # 4. Backward
        loss.backward()

        # 5. Update weights
        optimizer.step()

        total_loss += loss.item()

    if (epoch+1) % 200 == 0:
        print(f"Epoch {epoch+1}: Loss =
{total_loss/len(training_data):.4f}")

print("Training hoàn tất!")
```

→ **Output:**

```
--- BẮT ĐẦU TRAINING ---

Epoch 200: Loss = 1.5502

Epoch 400: Loss = 1.5386

Epoch 600: Loss = 1.5330

Epoch 800: Loss = 1.5292

Epoch 1000: Loss = 1.5263

Training hoàn tất!
```

```
# Lấy ma trận Embedding ra (đã train xong)
# W1 có shape [emb_dim, vocab_size] do cách PyTorch lưu Linear, nên ta cần
Transpose
# Tuy nhiên ở trên ta định nghĩa Linear(vocab, emb), nên weight có shape
[emb, vocab].
# Ta cần lấy Transpose để có shape [vocab, emb] -> Mỗi hàng là 1 sản phẩm.
final_embeddings = model.W1.weight.data.T

# --- HÀM TÍNH TOÁN ĐỘ TƯƠNG ĐỒNG (THỦ CÔNG) ---
```

```python
def get_cosine_similarity(vec_a, vec_b):
    # 1. Tích vô hướng (Dot product)
    dot_product = torch.dot(vec_a, vec_b)

    # 2. Độ dài vector (Norm) = căn bậc hai của tổng bình phương
    norm_a = torch.norm(vec_a)
    norm_b = torch.norm(vec_b)

    # 3. Cosine
    return dot_product / (norm_a * norm_b)

def recommend_manual(product_name, k=3):
    if product_name not in prod2idx:
        print("Sản phẩm không tồn tại!")
        return

    target_id = prod2idx[product_name]
    target_vec = final_embeddings[target_id]

    print(f"\n--- Đang tìm sản phẩm giống '{product_name}'... ---")
    print(f"Vector đặc trưng (đã học): {target_vec.numpy()}")

    similarities = []

    # Duyệt qua tất cả sản phẩm khác để tính khoảng cách
    for i in range(vocab_size):
        if i == target_id: continue # Bỏ qua chính nó

        other_vec = final_embeddings[i]
        score = get_cosine_similarity(target_vec, other_vec)

        similarities.append((idx2prod[i], score.item()))

    # Sắp xếp giảm dần theo điểm score
    similarities.sort(key=lambda x: x[1], reverse=True)

    # In kết quả top K
    for item, score in similarities[:k]:
        print(f"  > Gợi ý: {item} | Độ tương đồng: {score:.4f}")

# --- KIỂM TRA KẾT QUẢ ---
# Case 1: Mua iPhone (Kỳ vọng: Ốp lưng, Sạc nhanh, AirPods...)
recommend_manual('iPhone')

# Case 2: Mua Chuột (Kỳ vọng: Bàn phím, Lót chuột, Laptop...)
recommend_manual('Chuột')

# Case 3: Mua Laptop_Dell (Kỳ vọng: Chuột, Balo...)
recommend_manual('Laptop_Dell')
```

→ Output:

```
--- Đang tìm sản phẩm giống 'iPhone'... ---
```

```
Vector đặc trưng (đã học): [-0.42423773  1.104623    1.0103507  -0.01696407]
  > Gợi ý: Samsung | Độ tương đồng: 0.9178
  > Gợi ý: GalaxyBuds | Độ tương đồng: 0.6569
  > Gợi ý: Bàn_phím | Độ tương đồng: 0.3566

--- Đang tìm sản phẩm giống 'Chuột'... ---
Vector đặc trưng (đã học): [-0.05440662  0.9294625  -0.5109966  -0.99724704]
  > Gợi ý: Cường_lực | Độ tương đồng: 0.3732
  > Gợi ý: Ốp_lưng | Độ tương đồng: 0.2787
  > Gợi ý: Balo | Độ tương đồng: 0.2544

--- Đang tìm sản phẩm giống 'Laptop_Dell'... ---
Vector đặc trưng (đã học): [-1.1470968  -1.8891379  -1.1324849   0.23432276]
  > Gợi ý: Lót_chuột | Độ tương đồng: 0.8270
  > Gợi ý: AirPods | Độ tương đồng: 0.4898
  > Gợi ý: Hub_USB | Độ tương đồng: 0.4448
```

B. CBOW

```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

# --- 1. DỮ LIỆU & TỪ ĐIỂN (GIỮ NGUYÊN) ---
transactions = [
    ['iPhone', 'Ốp_lưng', 'Sạc_nhanh', 'AirPods'],
    ['Samsung', 'Ốp_lưng', 'Sạc_nhanh', 'GalaxyBuds'],
    ['Laptop_Dell', 'Chuột', 'Bàn_phím', 'Lót_chuột'],
    ['Macbook', 'Chuột', 'Hub_USB', 'Túi_chống_sốc'],
    ['iPhone', 'Sạc_nhanh', 'Cường_lực'],
    ['Samsung', 'Sạc_nhanh', 'Cường_lực'],
    ['Laptop_Dell', 'Chuột', 'Balo'],
    ['Bàn_phím', 'Chuột', 'Lót_chuột'],
    ['iPhone', 'AirPods', 'Apple_Watch']
]
```

```
products = sorted(list(set([item for sublist in transactions for item in
sublist])))
prod2idx = {p: i for i, p in enumerate(products)}
idx2prod = {i: p for p, i in prod2idx.items()}
vocab_size = len(products)
embedding_dim = 4

# --- 2. TẠO TRAINING DATA CHO CBOW ---
# Input: [Context_1, Context_2, ...] -> Output: Target
window_size = 2
cbow_train_data = []

for bill in transactions:
    indices = [prod2idx[p] for p in bill]
    for i, target_idx in enumerate(indices):
        context_idxs = []
        # Quét cửa sổ trái phải
        start = max(0, i - window_size)
        end = min(len(indices), i + window_size + 1)

        for j in range(start, end):
            if i != j: # Khác vị trí target
                context_idxs.append(indices[j])

        # Chỉ train nếu có context (tránh giỏ hàng chỉ có 1 món)
        if len(context_idxs) > 0:
            cbow_train_data.append((context_idxs, target_idx))

print(f"Số mẫu training CBOW: {len(cbow_train_data)}")
# Ví dụ: Context là [iPhone, Sạc_nhanh] -> Target là Ốp_lưng
print(f"Ví dụ mẫu đầu tiên: Context IDs {cbow_train_data[0][0]} -> Target
ID {cbow_train_data[0][1]}")
```

→ Output:

Số mẫu training CBOW: 31

Ví dụ mẫu đầu tiên: Context IDs [15, 12] -> Target ID 14

```
class ManualCBOW(nn.Module):
    def __init__(self, vocab_size, emb_dim):
        super(ManualCBOW, self).__init__()
        # W1: Ma trận Input -> Hidden (Embedding)
        self.W1 = nn.Linear(vocab_size, emb_dim, bias=False)
        # W2: Ma trận Hidden -> Output
        self.W2 = nn.Linear(emb_dim, vocab_size, bias=False)

    def forward(self, context_indices):
        # 1. TẠO VECTOR ĐẦU VÀO TỔNG HỢP (MANUAL MEAN)
        # Thay vì đưa vào 1 one-hot, ta tạo vector đại diện cho cả ngữ cảnh
        x_context_mean = torch.zeros(vocab_size)
```

```
        for idx in context_indices:
            x_one_hot = torch.zeros(vocab_size)
            x_one_hot[idx] = 1.0
            x_context_mean += x_one_hot # Cộng dồn

        # Chia trung bình (Mean)
        x_context_mean = x_context_mean / len(context_indices)

        # 2. Hidden Layer (Projection)
        # h = W1.T * x_mean
        hidden = self.W1(x_context_mean)

        # 3. Output Layer (Score)
        score = self.W2(hidden)

        return score

# Khởi tạo
model_cbow = ManualCBOW(vocab_size, embedding_dim)
optimizer = optim.SGD(model_cbow.parameters(), lr=0.05)
criterion = nn.CrossEntropyLoss() # Đã bao gồm Softmax

# --- 3. TRAINING LOOP ---
print("\n--- BẮT ĐẦU TRAIN CBOW ---")
for epoch in range(1000):
    total_loss = 0
    for contexts, target in cbow_train_data:
        optimizer.zero_grad()

        # Forward pass (Truyền list các index ngữ cảnh vào)
        pred_scores = model_cbow(contexts)

        # Loss
        loss = criterion(pred_scores.unsqueeze(0), torch.tensor([target]))

        loss.backward()
        optimizer.step()
        total_loss += loss.item()

    if (epoch+1) % 200 == 0:
        print(f"Epoch {epoch+1}: Loss =
 {total_loss/len(cbow_train_data):.4f}")
```

→ Output:

--- BẮT ĐẦU TRAIN CBOW ---

Epoch 200: Loss = 0.4520

Epoch 400: Loss = 0.3921

Epoch 600: Loss = 0.3793

```
Epoch 800: Loss = 0.3727

Epoch 1000: Loss = 0.3684
```

```python
# Lấy trọng số đã học
# Transpose lại vì nn.Linear lưu shape [out, in]
final_embeddings_cbow = model_cbow.W1.weight.data.T

# Hàm tính Cosine (Dùng lại hàm cũ)
def get_cosine_similarity(vec_a, vec_b):
    return torch.dot(vec_a, vec_b) / (torch.norm(vec_a) * torch.norm(vec_b))

def recommend_cbow(product_name):
    if product_name not in prod2idx: return
    target_id = prod2idx[product_name]
    target_vec = final_embeddings_cbow[target_id]

    similarities = []
    for i in range(vocab_size):
        if i == target_id: continue
        other_vec = final_embeddings_cbow[i]
        score = get_cosine_similarity(target_vec, other_vec)
        similarities.append((idx2prod[i], score.item()))

    similarities.sort(key=lambda x: x[1], reverse=True)
    print(f"\n[CBOW] Gợi ý cho '{product_name}':")
    for item, score in similarities[:3]:
        print(f"  - {item}: {score:.4f}")

# --- TEST KẾT QUẢ ---
# 1. iPhone -> Kỳ vọng: Ốp lưng, Sạc nhanh...
recommend_cbow('iPhone')

# 2. Laptop_Dell -> Kỳ vọng: Chuột, Balo...
recommend_cbow('Laptop_Dell')
```

→ Output:

```
[CBOW] Gợi ý cho 'iPhone':
  - Samsung: 0.9858
  - Ốp_lưng: 0.4130
  - GalaxyBuds: 0.3751
[CBOW] Gợi ý cho 'Laptop_Dell':
  - Túi_chống_sốc: 0.6279
  - Macbook: 0.6067
  - Lót_chuột: 0.5536
```

V. Summary

1. Motivation and Problem Framing

One-hot vectors are sparse, high-dimensional, and orthogonal — they waste memory and cannot capture semantic similarity. **Word embeddings** map tokens into dense, low-dimensional vectors so that **geometric proximity encodes meaning**. This enables analogy arithmetic (e.g., *king* − *man* + *woman* ≈ *queen*), improves downstream learning, and generalizes to non-text co-occurrence tasks (e.g., products in invoices).

2. Model Architectures

Skip-gram

- Predicts surrounding context words given a center word.
- Learns two matrices: input (center) embeddings and output (context) embeddings.
- Training pulls center vectors toward true context vectors and pushes them away from negatives.
- Strong for rare words but usually slower to train.

CBOW (Continuous Bag-of-Words)

- Predicts the center word from the average (mean) of surrounding context embeddings.
- Shares the same softmax/likelihood formulation but aggregates context before scoring.
- Typically faster and yields smoother vectors for frequent words.

Practical detail (shared)

- Both compute dot-product logits followed by softmax and minimize negative log-likelihood.
- For scalability, **Negative Sampling** (or Hierarchical Softmax) is used instead of full-vocabulary softmax.

3. Computation & Training Flow

For both architectures the training loop follows these steps:

1. **Embedding lookup** (center/context → vector).
2. **Hidden projection** (CBOW: mean of context vectors; Skip-gram: center vector).
3. **Dot-product logits** with output/context matrix.
4. **Softmax → class probabilities** (approximated by negative sampling in practice).
5. **Negative log-likelihood (cross-entropy) loss.**
6. **SGD (or Adam) updates** to input/output embedding matrices.

Key hyperparameters: embedding dimension (d), window size (m), negative samples (k), learning rate (η), number of epochs — these control convergence and semantic quality.

4. Implementation & Experiments (consistent with provided data)

- Didactic PyTorch implementations were provided for both Skip-gram and CBOW, including 2-D runs (d=2) for visualization and larger runs for product-invoice data (embedding_dim=4 in the invoice example).

- Data preprocessing: tokenization, vocabulary indexing, and creation of training pairs: (target → context) for Skip-gram, (contexts → target) for CBOW.

- Visualization: 2D scatter plots and cosine-similarity heatmaps to inspect clustering (gendered words, analogies).

- **Product-invoice use case:** treated products as "words" and baskets as "sentences"; trained Skip-gram/CBOW to produce item embeddings and ranked nearest neighbors by cosine similarity. Example outputs from the experiments (consistent with your run):

    - *iPhone* → **Samsung**, **cases**, **GalaxyBuds** (high cosine similarity)
    - *Laptop_Dell* → **shock-proof bag (Túi_chống_sốc)**, **Macbook**, **Lót_chuột**
      These results show that co-purchase semantics were successfully captured without manual rules.

5. Effectiveness & Model Comparison

- **Quality:** Cosine retrieval on trained embeddings surfaces meaningful top-k neighbors, confirming semantic structure.
- **Skip-gram vs CBOW:** Skip-gram better represents rare tokens; CBOW trains faster and performs well on frequent tokens. Choice depends on corpus size and token frequency distribution.

Aspect	**CBOW (Continuous Bag of Words)**	**Skip-Gram**
Concept	Predicts a target word based on context words.	Predicts context words given a target word.
Context Window	Typically smaller (2-5 words)	Can handle larger windows (5-20 words)
Training Process	Minimizes cross-entropy loss to predict the target word.	Maximizes the likelihood of context words around a target word using techniques like negative sampling or

		hierarchical softmax.
Training Speed	Faster (single prediction per context window)	Slower (multiple predictions per target word)
Performance	Better for frequent words, syntactic relationship	Better for rare words, semantic relationships.
Overfitting	Can overfit frequent words	Less prone to overfitting frequent words
Model Size	Smaller	Larger
Data Requirements	Needs less data	Needs more data, works well with large datasets
Use Cases	Suitable for tasks requiring speed over detailed word representations, like text classification and sentiment analysis.	Ideal for tasks needing high-quality embeddings and detailed semantic relationships, such as word similarity tasks, named entity recognition, and machine translation.

6. Takeaways & Limitations

Takeaways:

- Embeddings convert unstructured tokens into dense vectors that preserve semantics and support analogy, retrieval, and recommendation tasks.
- The same methodology generalizes beyond words (products, users, events).

Limitations:

- Requires sufficient data to stabilize vectors.
- Poor at modeling **polysemy** (multiple senses) and **subword morphology** unless extended.
- Context-insensitive: static embeddings do not change with sentence context.

7. Recommendations & Possible Improvements

- For industrial scale: increase embedding dimension (≥50–300), use negative sampling, and train on much larger corpora.
- To handle rare forms and morphology: adopt **fastText** (subword-aware).

- To model context and polysemy: use **contextual encoders** (BERT/ELMo) or fine-tune transformer embeddings for downstream tasks.
- For production recommendations: combine learned embeddings with collaborative or content-based signals and evaluate with offline metrics (precision@k, recall@k, MAP).

VI. Exercises

1. What is the computational complexity for calculating each gradient? What could be the issue if the dictionary size is huge?
2. Some fixed phrases in English consist of multiple words, such as "new york". How to train their word vectors? Hint: see Section 4 in the word2vec paper Mikolov.Sutskever.Chen.ea.2013.
3. Let's reflect on the word2vec design by taking the skip-gram model as an example. What is the relationship between the dot product of two word vectors in the skip-gram model and the cosine similarity? For a pair of words with similar semantics, why may the cosine similarity of their word vectors (trained by the skip-gram model) be high?

Solutions

The end of file:

https://colab.research.google.com/drive/1AqaSKrozlrNKx-6tEISOHcSbsOGas38S?authuser=1

Exercise 1: Computational Complexity and Dictionary Size Issues

Question: What is the computational complexity for calculating each gradient? What could be the issue if the dictionary size is huge?

Computational Complexity Analysis

Looking at the gradient formula for Skip-gram (from the theory section):

$$\frac{\partial log P(w_o|w_c)}{\partial v_c} = u_o - \sum_{j \in V} P(w_j|w_c) u_j$$

For Skip-Gram:

To compute the gradient for ONE center-context pair, we need to:

1. **Calculate the softmax denominator**: $\sum_{i \in V} exp(u_i^\top v_c)$
    - This requires computing dot products for ALL vocabulary words: $|V|$ dot products
    - Each dot product is $O(d)$ where d is embedding dimension
    - Total: $O(|V| \cdot d)$

2. **Compute the weighted sum**: $\sum_{j \in V} P(w_j|w_c)u_j$
    - Requires iterating over all $|V|$ words
    - Each operation is $O(d)$
    - Total: $O(|V| \cdot d)$

Total complexity per gradient: $O(|V| \cdot d)$

For CBOW: Similar analysis gives the same complexity: $O(|V| \cdot d)$

Issues with Huge Dictionary Size

When the vocabulary size is huge (e.g., 100K+ words), several critical problems emerge:

1. **Computational Bottleneck**:
    - Computing softmax over 100K words for every training example becomes prohibitively expensive
    - Example: With |V| = 100,000 and d = 300, each gradient requires ~30 million operations
    - For a corpus with millions of training examples, this becomes intractable
2. **Memory Requirements**:
    - Storing $|V| \times d$ parameters requires significant memory
    - Example: 100K words × 300 dimensions × 4 bytes (float32) = 120 MB just for one embedding matrix
    - With two matrices (input and output), this doubles
3. **Training Speed**:
    - Each gradient update requires $O(|V|)$ operations
    - Training becomes extremely slow, making it impractical for large-scale applications

Solutions Used in Practice

1. Negative Sampling:

Instead of computing softmax over the entire vocabulary, we sample K negative examples (typically K = 5-20).

How it works:

- For each positive example (center word, context word), we:
    1. Keep the positive pair

2. Sample K random words from the vocabulary as "negative" examples
3. Train a binary classifier: positive = 1, negatives = 0

Complexity reduction:

- Original: $O(|V| \cdot d)$
- With Negative Sampling: $O(K \cdot d)$ where K << |V|
- Example: With K=5, complexity reduces from 30M to 1,500 operations (20,000× speedup!)

Implementation in gensim:

```
from gensim.models import Word2Vec

# Negative sampling is used by default
model = Word2Vec(sentences,
                 vector_size=100,
                 window=5,
                 negative=5,      # Sample 5 negative examples
                 min_count=1)
```

2. Hierarchical Softmax:

Use a binary tree structure (Huffman tree) where each word is a leaf node.

How it works:

- Organize vocabulary as a binary tree
- Each word only needs to traverse the path from root to its leaf
- Path length = $O(log|V|)$ instead of $O(|V|)$

Complexity reduction:

- Original: $O(|V| \cdot d)$
- With Hierarchical Softmax: $O(log|V| \cdot d)$
- Example: With |V| = 100,000, log(100,000) ≈ 17, so complexity reduces from 30M to ~5,100 operations

Implementation:

```
# Use hierarchical softmax instead of negative sampling
model = Word2Vec(sentences,
                 vector_size=100,
                 window=5,
                 hs=1,           # Enable hierarchical softmax
                 negative=0)     # Disable negative sampling
```

3. Subsampling Frequent Words:

Additionally, we can subsample very frequent words (like "the", "a", "is") to speed up training and improve quality.

Why it helps:

- Frequent words provide less information (appear everywhere)
- Reducing their frequency in training data speeds up learning
- Improves representation quality for rare but informative words

Both Negative Sampling and Hierarchical Softmax maintain the quality of learned embeddings while dramatically improving training efficiency. In practice, Negative Sampling is more commonly used due to its simplicity and effectiveness.

Exercise 2: Training Word Vectors for Multi-Word Phrases

Question: Some fixed phrases in English consist of multiple words, such as "new york". How to train their word vectors?

Problem Statement

Multi-word phrases (also called n-grams or collocations) like "New York", "machine learning", or "ice cream" should be treated as single tokens because:

- **Their meaning is NOT compositional**: "New York" ≠ "new" + "york"
- **They represent distinct semantic concepts**: "New York" is a city name, not an adjective + noun
- **Treating them separately loses information**: The model cannot learn that "New York" is semantically closer to "city" than to "new" or "york" individually

Solution Approach (from word2vec paper Section 4)

The paper proposes using a **statistical phrase detection** method to automatically identify and merge frequent word pairs.

1. Phrase Detection Using Statistical Scoring:

The paper uses a scoring function to identify phrases:

$$score(w_i, w_j) = \frac{count(w_i w_j) - \delta}{count(w_i) \times count(w_j)}$$

Where:

- $count(w_i w_j)$ = number of times words w_i and w_j appear together (as a bigram)
- $count(w_i)$ = frequency of word w_i alone (as a unigram)
- δ = discounting coefficient (typically 5-10) that prevents rare word pairs from forming phrases

Intuition:

- **High score** means the words appear together much more than expected by chance → likely a phrase
- **Low score** means the words appear together only by chance → not a phrase

Example Calculation:

Suppose in our corpus:

- "new" appears 1000 times
- "york" appears 500 times
- "new york" appears together 400 times
- $\delta = 5$

$$score(new, york) = \frac{400-5}{1000 \times 500} = \frac{395}{500{,}000} = 0.00079$$

Now compare with "new car":

- "new car" appears together 10 times

$$score(new, car) = \frac{10-5}{1000 \times 300} = \frac{5}{300{,}000} = 0.000017$$

The much higher score for "new york" indicates it's a phrase!

2. Training Process (Iterative):

The process involves multiple passes:

First Pass - Bigram Detection:

1. Scan corpus and compute scores for all adjacent word pairs
2. If score > threshold, merge into single token
3. Example: "New" + "York" → "New_York"

Second Pass - Trigram Detection (Optional):

1. Repeat phrase detection on the transformed corpus
2. Can find longer phrases: "New_York" + "City" → "New_York_City"

Final Pass - Train Word2Vec:

1. Train skip-gram or CBOW on the corpus with phrases treated as single words
2. "New_York" gets its own embedding vector, separate from "new" and "york"

Complete Implementation Example

```
from gensim.models import Word2Vec
from gensim.models.phrases import Phrases, Phraser
```

```
# Sample corpus with multi-word phrases
sentences = [
    ['new', 'york', 'is', 'a', 'city'],
    ['new', 'york', 'city', 'is', 'large'],
    ['i', 'love', 'new', 'york'],
    ['machine', 'learning', 'is', 'fun'],
    ['i', 'study', 'machine', 'learning'],
    ['deep', 'learning', 'uses', 'neural', 'networks'],
    ['ice', 'cream', 'is', 'cold'],
    ['i', 'eat', 'ice', 'cream']
]

print("=== BEFORE PHRASE DETECTION ===")
print("Sample sentence:", sentences[0])
# Output: ['new', 'york', 'is', 'a', 'city']

# STEP 1: Train phrase detector (bigrams)
# min_count=2: phrase must appear at least 2 times
# threshold=1: lower threshold = more phrases detected (default is 10)
phrase_model = Phrases(sentences, min_count=2, threshold=1)

# Apply phrase detection to transform corpus
sentences_with_phrases = [phrase_model[sent] for sent in sentences]

print("\n=== AFTER PHRASE DETECTION (First Pass) ===")
for i, sent in enumerate(sentences_with_phrases[:5]):
    print(f"Sentence {i+1}: {sent}")
# Output:
# Sentence 1: ['new_york', 'is', 'a', 'city']
# Sentence 2: ['new_york', 'city', 'is', 'large']
# Sentence 3: ['i', 'love', 'new_york']
# Sentence 4: ['machine_learning', 'is', 'fun']
# Sentence 5: ['i', 'study', 'machine_learning']

# STEP 2: Optional - Second pass for longer phrases (trigrams)
phrase_model_2 = Phrases(sentences_with_phrases, min_count=2, threshold=1)
sentences_final = [phrase_model_2[sent] for sent in
sentences_with_phrases]

print("\n=== AFTER SECOND PASS (Trigrams) ===")
# May find: "new_york_city" if it appears frequently enough

# STEP 3: Train Word2Vec on phrase-enhanced corpus
model = Word2Vec(sentences_final,
                 vector_size=10,
                 window=2,
                 min_count=1,
                 epochs=100)

print("\n=== TRAINED VOCABULARY (includes phrases) ===")
print("Vocabulary:", list(model.wv.key_to_index.keys()))
# Output includes: 'new_york', 'machine_learning', 'ice_cream', etc.
```

```
print("\n=== FINDING SIMILAR WORDS/PHRASES ===")
print("✓ 'new_york' is treated as a single token!")
print("  Similar to 'new_york':", model.wv.most_similar('new_york',
topn=3))
# Output: [('city', 0.76), ('a', 0.30), ...]
```

What the Code Does:

1. **Phrases(sentences, ...)**:
    - Scans all sentences to count bigram frequencies
    - Computes phrase scores using the formula above
    - Identifies bigrams with scores above the threshold
2. **phrase_model[sent]**:
    - Transforms a sentence by merging detected phrases
    - "new york" → "new_york" (underscore indicates it's a phrase)
3. **Word2Vec(sentences_final, ...)**:
    - Trains embeddings on the phrase-enhanced corpus
    - "new_york" gets its own vector, separate from "new" and "york"

Why It Works

- **Non-compositional phrases** get dedicated embeddings that capture their unique meaning
- The model learns that "new_york" is semantically closer to "city" than to "new" or "york" individually
- This improves performance on tasks requiring phrase-level understanding
- The statistical scoring automatically identifies phrases without manual annotation

Real-world Impact:

In practice, phrase detection significantly improves word embeddings for:

- Named entities: "New York", "United States", "Machine Learning"
- Technical terms: "neural networks", "deep learning", "natural language processing"
- Idiomatic expressions: "ice cream", "hot dog", "break down"

Exercise 3: Dot Product, Cosine Similarity, and Semantic Relationships

Question: What is the relationship between the dot product of two word vectors in the skip-gram model and the cosine similarity? For a pair of words with similar semantics, why may the cosine similarity of their word vectors (trained by the skip-gram model) be high?

Part 1: Relationship Between Dot Product and Cosine Similarity

The dot product of two word vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ is mathematically related to cosine similarity as follows:

$$v_1 \cdot v_2 = ||v_1|| \cdot ||v_2|| \cdot cos(\theta)$$

Where:

- $v_1 \cdot v_2$ = dot product
- $||v_1||$ and $||v_2||$ = magnitudes (norms) of the vectors
- θ = angle between the vectors
- $cos(\theta)$ = cosine similarity

Derivation:

From the definition of dot product:

$$v_1 \cdot v_2 = \sum_{i=1}^{d} v_{1i} \cdot v_{2i}$$

And from the geometric interpretation:

$$v_1 \cdot v_2 = ||v_1|| \cdot ||v_2|| \cdot cos(\theta)$$

Therefore:

$$cos(\theta) = \frac{v_1 \cdot v_2}{||v_1|| \cdot ||v_2||}$$

Key Differences:

1. **Dot Product**:
    - **Depends on both magnitude and direction**
    - Used in softmax during training: $P(w|context) \propto exp(u_w \cdot v_c)$
    - Larger magnitude vectors produce larger dot products
    - Range: $(-\infty, +\infty)$
    - Example: If $v_1 = [2, 0]$ and $v_2 = [3, 0]$, dot product = 6
2. **Cosine Similarity**:
    - **Measures pure directional similarity (normalized)**
    - Range: $[-1, 1]$, where 1 means identical direction, -1 means opposite
    - **Independent of vector magnitude** - only cares about angle
    - Better for semantic similarity tasks

- Example: Same vectors as above, cosine = 1.0 (both point in same direction)

Why Skip-gram Uses Dot Product (Not Cosine) in Training:

The training objective maximizes:

$$logP(w_{context}|w_{center}) = u_{context}^{\top} v_{center} - log\sum_{i} exp(u_i^{\top} v_{center})$$

The dot product is used because:

1. **Computational efficiency**: It's the natural form in the softmax formulation
2. **Magnitude encodes information**: Rare words may have smaller magnitude, frequent words larger
3. **Normalization would lose information**: Converting to unit vectors (cosine) would discard frequency signals
4. **Training stability**: Dot products work better with gradient-based optimization

However, for **similarity search** after training, we use **cosine similarity** because we care about semantic direction, not magnitude.

Part 2: Why Semantically Similar Words Have High Cosine Similarity

The Core Mechanism: Distributional Hypothesis

The fundamental principle is: **"Words that occur in similar contexts tend to have similar meanings"**

Example:

- "king" appears with context words: {crown, throne, royal, palace, kingdom, rules, power}
- "queen" appears with context words: {crown, throne, royal, palace, kingdom, rules, grace}
- They share similar context words! This is the key insight.

Step-by-Step Training Process:

During training, the Skip-gram model maximizes:

$$logP(w_{context}|w_{center}) = u_{context}^{\top} v_{center} - log\sum_{i} exp(u_i^{\top} v_{center})$$

What happens for "king":

- When "king" appears with "crown" nearby, the model increases $u_{crown}^{\top} v_{king}$
- When "king" appears with "throne" nearby, the model increases $u_{throne}^{\top} v_{king}$

- When "king" appears with "royal" nearby, the model increases $u_{royal}^{\top} v_{king}$

What happens for "queen":

- When "queen" appears with "crown" nearby, the model increases $u_{crown}^{\top} v_{queen}$ (same context word!)
- When "queen" appears with "throne" nearby, the model increases $u_{throne}^{\top} v_{queen}$ (same context word!)
- When "queen" appears with "royal" nearby, the model increases $u_{royal}^{\top} v_{queen}$ (same context word!)

Vector Alignment Mechanism:

Since both "king" and "queen" need to have **high dot products with the SAME context words**, their center vectors v_{king} and v_{queen} must point in **similar directions**!

Visual Intuition:

```
Before Training (Random):
  king vector:  [0.2, -0.1]  (random direction)
  queen vector: [-0.3, 0.4]  (different random direction)
  Cosine similarity: 0.13 (low)

After Training:
  Both vectors are "pulled" toward the same context words:
  - Both align with crown vector [0.8, 0.6]
  - Both align with throne vector [0.9, 0.5]
  - Both align with royal vector [0.7, 0.7]

  king vector:  [0.82, 0.58]  (aligned with contexts)
  queen vector: [0.80, 0.60]  (aligned with same contexts)
  Cosine similarity: 0.998 (high!)
```

Mathematical Proof:

Suppose two words w_a and w_b share many context words. Let C be the set of common context words.

Objective Function (what we're maximizing):

For word w_a:

$$\sum_{w_c \in C} logP(w_c|w_a) = \sum_{w_c \in C} \left[u_c^{\top} v_a - logZ_a \right]$$

For word w_b:

$$\sum_{w_c \in C} logP(w_c|w_b) = \sum_{w_c \in C} \left[u_c^{\top} v_b - logZ_b \right]$$

To maximize both, we need:

- $u_c^\top v_a$ to be large for all $c \in C$
- $u_c^\top v_b$ to be large for all $c \in C$

This means:

- v_a must align with all u_c (point in similar direction)
- v_b must align with all u_c (point in similar direction)

Therefore: v_a and v_b must point in similar directions!

Result: $cos(v_a, v_b) \approx 1$ (high cosine similarity)

Empirical Verification with Code

Let's verify this with a practical example:

```
from gensim.models import Word2Vec
import numpy as np

# Sample corpus with clear semantic patterns
corpus_demo = [
    "king rules the kingdom with power",
    "queen rules the kingdom with grace",
    "the king sits on his throne",
    "the queen sits on her throne",
    "king wears a crown and royal robes",
    "queen wears a crown and royal robes",
    "prince will become king someday",
    "princess will become queen someday",
    "man works in the office building",
    "woman works in the office building",
    "boy plays football with friends",
    "girl plays football with friends"
] * 50  # Replicate for better training

# Tokenize
sentences_demo = [sent.split() for sent in corpus_demo]

# Train model
model_demo = Word2Vec(sentences_demo, vector_size=50, window=3,
                      min_count=1, epochs=100, sg=1, seed=42)

# Function to analyze similarity
def analyze_similarity(word1, word2, model):
    vec1 = model.wv[word1]
    vec2 = model.wv[word2]

    # Compute dot product
```

```
    dot_prod = np.dot(vec1, vec2)

    # Compute norms
    norm1 = np.linalg.norm(vec1)
    norm2 = np.linalg.norm(vec2)

    # Compute cosine similarity manually
    cos_sim = dot_prod / (norm1 * norm2)

    # Verify with gensim's built-in
    gensim_sim = model.wv.similarity(word1, word2)

    print(f"\n'{word1}' vs '{word2}':")
    print(f"  Dot Product:       {dot_prod:.4f}")
    print(f"  ||{word1}||:          {norm1:.4f}")
    print(f"  ||{word2}||:         {norm2:.4f}")
    print(f"  Cosine Similarity: {cos_sim:.4f}")
    print(f"  Gensim Similarity:  {gensim_sim:.4f} (verification)")

print("=" * 70)
print("EXERCISE 3 VERIFICATION: Cosine Similarity Analysis")
print("=" * 70)

print("\n[1] SEMANTICALLY SIMILAR PAIRS (Should have HIGH cosine
similarity)")
print("-" * 70)
analyze_similarity('king', 'queen', model_demo)
analyze_similarity('man', 'woman', model_demo)
analyze_similarity('prince', 'princess', model_demo)

print("\n\n[2] SEMANTICALLY DIFFERENT PAIRS (Should have LOW cosine
similarity)")
print("-" * 70)
analyze_similarity('king', 'building', model_demo)
analyze_similarity('crown', 'football', model_demo)

print("\n\n[3] FAMOUS ANALOGY TEST: king - man + woman ≈ queen")
print("-" * 70)
result = model_demo.wv.most_similar(
    positive=['king', 'woman'],
    negative=['man'],
    topn=5
)
print("  Analogy: king - man + woman =")
for word, score in result:
    marker = "✓✓✓" if word == 'queen' else ""
    print(f"    {word}: {score:.4f} {marker}")
```

Expected Output:

```
[1] SEMANTICALLY SIMILAR PAIRS
'king' vs 'queen':
  Dot Product:       5.7316
  ||king||:          2.4970
```

```
  ||queen||:         2.5097
  Cosine Similarity: 0.9146  ← HIGH!

'man' vs 'woman':
  Cosine Similarity: 0.9988  ← VERY HIGH!

[2] SEMANTICALLY DIFFERENT PAIRS
'king' vs 'building':
  Cosine Similarity: 0.0988  ← LOW!

[3] ANALOGY TEST
  Analogy: king - man + woman =
    queen: 0.9136 ✓✓✓  ← Correct!
```

What This Demonstrates:

1. **Semantically similar words** (king/queen, man/woman) have **HIGH cosine similarity** (0.91-0.99)
2. **Semantically different words** (king/building) have **LOW cosine similarity** (0.10)
3. **Vector arithmetic works**: king - man + woman ≈ queen, proving the geometric structure captures semantic relationships

Key Takeaway:

The model **never explicitly learns** that "king and queen are similar"—it **discovers this automatically** from their shared contexts. This is the power of **self-supervised learning** from distributional patterns: **semantic relationships emerge as geometric relationships in vector space!**

The magic is that by simply trying to predict context words, the model learns to organize words in a geometric space where:

- Similar meanings → Similar directions → High cosine similarity
- Different meanings → Different directions → Low cosine similarity

This is why word embeddings are so powerful: they transform linguistic relationships into mathematical relationships that computers can efficiently work with.

References

A. Zhang *et al.*, "15.1. Word Embedding (word2vec)," in *Dive into Deep Learning*, 2021. [Online]. Available: https://d2l.ai/chapter_natural-language-processing-pretraining/word2vec.html. [Accessed: Dec. 06, 2025].

T. Vu, "Word2vec," in *Machine Learning cho dữ liệu dạng bảng*, 2021. [Online]. Available: https://machinelearningcoban.com/tabml_book/ch_embedding/word2vec.html. [Accessed: Dec. 06, 2025].

R. Singh, "Learning Word Embeddings with CBOW and Skip-gram," *Medium*, Oct. 11, 2024. [Online]. Available: https://medium.com/@RobuRishabh/learning-word-embeddings-with-cbow-and-skip-gram-b834bde18de4. [Accessed: Dec. 06, 2025].

GeeksforGeeks, "Implement your own word2vec(skip-gram) model in Python," *GeeksforGeeks*, Jul. 11, 2025. [Online]. Available: https://www.geeksforgeeks.org/python/implement-your-own-word2vecskip-gram-model-in-python/. [Accessed: Dec. 06, 2025].

M. Riva, "Word Embeddings: CBOW vs Skip-Gram," *Baeldung*, Feb. 13, 2025. [Online]. Available: https://www.baeldung.com/cs/word-embeddings-cbow-vs-skip-gram. [Accessed: Dec. 06, 2025].