

```

import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from collections import Counter
from torch.utils.data import Dataset, DataLoader

# Hyperparameters
embedding_dim = 100
context_size = 2 # Number of context words to use
num_negative_samples = 5 # Number of negative samples per positive sample
learning_rate = 0.001
num_epochs = 5

# Example corpus
corpus = [
    "we are what we repeatedly do excellence then is not an act but a habit",
    "the only way to do great work is to love what you do",
    "if you can dream it you can do it",
    "do not wait to strike till the iron is hot but make it hot by striking",
    "whether you think you can or you think you cannot you are right",
]

```

```

# -----
# Function: preprocess_corpus
# Purpose: Prepare the text corpus for Word2Vec training by
#           - Tokenizing sentences into words
#           - Building a vocabulary
#           - Creating index mappings (word <-> integer ID)
# -----

def preprocess_corpus(corpus):
    """
    corpus : list of strings
        Example: ["he is a king", "she is a queen"]

    Returns:
        words      -> list of all words in the corpus (with repetition)
        word_to_idx -> dictionary mapping each unique word to an integer index
        idx_to_word -> reverse dictionary mapping each index to its word
    """

    # 🌱 1. Flatten all sentences into a single list of words
    # 'sentence.split()' splits each sentence into words
    # The nested loop [word for sentence in corpus for word in sentence.split()]
    # means: for each sentence in corpus, for each word in that sentence, collect it.
    words = [word for sentence in corpus for word in sentence.split()]

    # 🍷 Example result:
    # words = ['he', 'is', 'a', 'king', 'she', 'is', 'a', 'queen', ...]

    # 🌱 2. Create the vocabulary (unique set of words)
    # 'set(words)' removes duplicates → gives all distinct words in the corpus
    vocab = set(words)

    # 🌱 3. Map each unique word to a unique integer index
    # enumerate(vocab) gives (index, word) pairs like (0, 'he'), (1, 'queen'), ...
    # we store them in a dictionary: {word: index}
    word_to_idx = {word: idx for idx, word in enumerate(vocab)}

    # 🌱 4. Create the reverse mapping
    # This allows us to convert indices back to words
    idx_to_word = {idx: word for word, idx in word_to_idx.items()}

    # 🌱 5. Return all 3 structures
    # 'words' → full tokenized list (for training)
    # 'word_to_idx' → used to convert words into numeric form
    # 'idx_to_word' → used to decode numeric predictions back to words
    return words, word_to_idx, idx_to_word

# -----
# Example usage:
# -----

```

```
# Call the preprocessing function
words, word_to_idx, idx_to_word = preprocess_corpus(corpus)

# Now:
# words      -> ['he', 'is', 'a', 'king', 'she', 'is', 'a', 'queen', ...]
# word_to_idx -> {'he': 0, 'is': 1, 'a': 2, 'king': 3, ...}
# idx_to_word -> {0: 'he', 1: 'is', 2: 'a', 3: 'king', ...}
```

words

```
'is',
'not',
'an',
'act',
'but',
'a',
'habit',
'the',
'only',
'way',
'to',
'do',
'great',
'work',
'is',
'to',
'love',
'what',
'you',
'do',
'if',
'you',
'can',
'dream',
'it',
'you',
'can',
'do',
'it',
'do',
'not',
'wait',
'to',
'strike',
'till',
'the',
'iron',
'is',
'hot',
'but',
'make',
'it',
'hot',
'by',
'striking',
'whether',
'you',
'think',
'you',
'can',
'can',
'or',
'you',
'think',
'you',
'cannot',
'you',
'are',
'right']
```

word\_to\_idx

```
{ 'wait': 0,
  'can': 1,
  'till': 2,
  'the': 3,
  'do': 4,
  'love': 5,
  'whether': 6,
  'great': 7,
  'it': 8,
  'strike': 9,
```

```
'hot': 10,
'then': 11,
'cannot': 12,
'think': 13,
'are': 14,
'by': 15,
'to': 16,
'only': 17,
'not': 18,
'or': 19,
'is': 20,
'we': 21,
'way': 22,
'habit': 23,
'an': 24,
'make': 25,
'dream': 26,
'iron': 27,
'you': 28,
'a': 29,
'what': 30,
'striking': 31,
'but': 32,
'work': 33,
'if': 34,
'excellence': 35,
'act': 36,
'right': 37,
'repeatedly': 38}
```

idx\_to\_word

```
{0: 'wait',
1: 'can',
2: 'till',
3: 'the',
4: 'do',
5: 'love',
6: 'whether',
7: 'great',
8: 'it',
9: 'strike',
10: 'hot',
11: 'then',
12: 'cannot',
13: 'think',
14: 'are',
15: 'by',
16: 'to',
17: 'only',
18: 'not',
19: 'or',
20: 'is',
21: 'we',
22: 'way',
23: 'habit',
24: 'an',
25: 'make',
26: 'dream',
27: 'iron',
28: 'you',
29: 'a',
30: 'what',
31: 'striking',
32: 'but',
33: 'work',
34: 'if',
35: 'excellence',
36: 'act',
37: 'right',
38: 'repeatedly'}
```

```
# -----
# FUNCTION: generate_training_data
# Purpose:
#   Create pairs of (target_word, context_word) from a list
#   of words, according to a given context window size.
# -----

def generate_training_data(words, word_to_idx, context_size):
    # Initialize an empty list to hold all training pairs
```

```

# Each item in 'data' will be a tuple: (target_word_index, context_word_index)
data = []

# Loop through each word in the corpus – but skip the first and last
# 'context_size' words to avoid index out-of-range errors
for i in range(context_size, len(words) - context_size):
    # -----
    # Step 1: Identify the 'target word'
    # -----
    # The word at position 'i' is our center (target) word.
    # We convert it to its integer index using 'word_to_idx'.
    target_word = word_to_idx[words[i]]

    # -----
    # Step 2: Collect 'context words' around the target
    # -----
    # Left-side context words: words[i-1], words[i-2], ... up to context_size
    # We move backwards using 'i - j - 1'
    left_context = [word_to_idx[words[i - j - 1]] for j in range(context_size)]

    # Right-side context words: words[i+1], words[i+2], ... up to context_size
    # We move forwards using 'i + j + 1'
    right_context = [word_to_idx[words[i + j + 1]] for j in range(context_size)]

    # Combine both sides into a single list of all context word indices
    context_words = left_context + right_context

    # -----
    # Step 3: Create training pairs
    # -----
    # For each context word, we make a (target, context) pair
    for context_word in context_words:
        data.append((target_word, context_word))

# Return the full list of training pairs
return data

# -----
# Example of using the function:
# -----
# words          → flat list of all words from the corpus
# word_to_idx    → mapping of each word to a unique index
# context_size   → number of words to take on each side
# -----

training_data = generate_training_data(words, word_to_idx, context_size)

```

```
training_data
```

```
(28, 1),
(28, 19),
(1, 28),
(1, 13),
(1, 19),
(1, 28),
(19, 1),
(19, 28),
(19, 28),
(19, 13),
(28, 19),
(28, 1),
(28, 13),
(28, 28),
(13, 28),
(13, 19),
(13, 28),
(13, 12),
(28, 13),
(28, 28),
(28, 12),
(28, 28),
(12, 28),
(12, 13),
(12, 28),
(12, 14),
(28, 12),
(28, 28),
(28, 14),
(28, 37)]
```

```
# -----
# IMPORTS
# -----
# These classes come from PyTorch – they are used to handle
# datasets and batching during model training.
from torch.utils.data import Dataset, DataLoader

# -----
# DEFINE CUSTOM DATASET CLASS
# -----
# This class wraps your (target, context) training pairs into
# a structure compatible with PyTorch's DataLoader.
# -----

class Word2VecDataset(Dataset): # Inherit from PyTorch's base Dataset class
    def __init__(self, data):
        """
        Constructor: called when you create a new dataset object.
        'data' is expected to be a list of tuples (target, context).
        Example: [(2, 5), (1, 4), (3, 2), ...]
        """
        self.data = data # Store the input data inside the object

    def __len__(self):
        """
        Returns the total number of samples in the dataset.
        This is required by PyTorch so it knows the dataset size.
        """
        return len(self.data)

    def __getitem__(self, idx):
        """
        Returns a single sample (target, context) pair at the given index.
        This is called automatically when the DataLoader fetches a batch.
        """
        return self.data[idx]

# -----
# CREATE DATASET AND DATALOADER OBJECTS
# -----

# Wrap your training data (list of tuples) inside the custom dataset
dataset = Word2VecDataset(training_data)

# Use PyTorch's DataLoader to:
# - Automatically divide data into batches
# - Shuffle the data for better training
```

```
# - Feed it to the model during each training iteration
dataloader = DataLoader(dataset, batch_size=32, shuffle=True)
```

dataloader

<torch.utils.data.dataloader.DataLoader at 0x7f6d5656a8d0>

```
# -----
# FUNCTION: get_negative_samples
# -----
# Purpose:
#   Generate random word indices (negative samples)
#   that act as "noise words" – words *not* related to
#   the target word in the current training pair.
#
# Why:
#   Instead of updating weights for *every* word in the vocabulary,
#   we update only for:
#   - The real positive context word
#   - A few randomly chosen "negative" words
#   → This makes training thousands of times faster.
# -----

import numpy as np # NumPy for random number generation

def get_negative_samples(target, num_negative_samples, vocab_size):
    # Create an empty list to store negative sample indices
    neg_samples = []

    # Keep generating random indices until we have enough negatives
    while len(neg_samples) < num_negative_samples:
        # Randomly pick an integer between 0 and (vocab_size - 1)
        # Each integer corresponds to a word in the vocabulary.
        neg_sample = np.random.randint(0, vocab_size)

        # Avoid sampling the target word itself
        # because the negative words must be *different* from the true target.
        if neg_sample != target:
            neg_samples.append(neg_sample)

    # Return the list of negative sample word indices
    return neg_samples
```

```
import torch
import torch.nn as nn

# -----
# CLASS: SkipGramNegSampling
# -----
# Implements the Skip-gram model with Negative Sampling
# using two embedding matrices:
#   - self.embeddings: for target (center) words
#   - self.context_embeddings: for context (neighbor) words
# -----

class SkipGramNegSampling(nn.Module):
    def __init__(self, vocab_size, embedding_dim):
        super(SkipGramNegSampling, self).__init__()

        # Total number of words in the vocabulary
        self.vocab_size = vocab_size

        # Embedding layer for target (center) words
        # Each word index maps to a dense vector of length embedding_dim
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)

        # Embedding layer for context words
        # 💡 At the end, the embeddings from these two layers are often averaged or reused as the final word vectors.
        # Having separate weights allows different representations
        self.context_embeddings = nn.Embedding(vocab_size, embedding_dim)

        # Log-sigmoid activation function
        # log(σ(x)) = log(1 / (1 + exp(-x)))
```

```

# Used for numerical stability in negative sampling loss
self.log_sigmoid = nn.LogSigmoid()

# -----
# FORWARD METHOD
# -----
# Inputs:
#   target → tensor of word indices for the center words
#   context → tensor of word indices for the positive context words
#   negative_samples → tensor of indices for sampled noise words
#
# Output:
#   loss value (scalar)
# -----
def forward(self, target, context, negative_samples):

    # -----
    # Step 1: Get embeddings for all inputs
    # -----

    # Embedding lookup for target (center) words
    # Shape: [batch_size, embedding_dim]
    target_embedding = self.embeddings(target)

    # Embedding lookup for context (true positive) words
    # Shape: [batch_size, embedding_dim]
    context_embedding = self.context_embeddings(context)

    # Embedding lookup for negative (noise) words
    # Shape: [batch_size, num_neg_samples, embedding_dim]
    negative_embeddings = self.context_embeddings(negative_samples)

    # -----
    # Step 2: Compute scores for positive pairs
    # -----
    # Dot product between target and context embeddings:
    #   s_pos = target_embedding · context_embedding
    # Then apply log-sigmoid to get  $\log(\sigma(s_{\text{pos}}))$ 
    # Shape: [batch_size]
    positive_score = self.log_sigmoid(
        torch.sum(target_embedding * context_embedding, dim=1)
    )

    # -----
    # Step 3: Compute scores for negative pairs
    # -----
    # Compute dot products between target embeddings and
    # each negative context word.
    #
    #   s_neg = - (target_embedding · neg_embedding)
    #
    # torch.bmm performs batch matrix multiplication:
    #   negative_embeddings: [batch, num_neg, embed_dim]
    #   target_embedding.unsqueeze(2): [batch, embed_dim, 1]
    # Result: [batch, num_neg, 1]
    #
    # After squeezing and summing across negative samples,
    # we get the total negative log probability per batch.
    negative_score = self.log_sigmoid(
        -torch.bmm(negative_embeddings, target_embedding.unsqueeze(2))
        .squeeze(2)
    ).sum(1)

    # -----
    # Step 4: Combine and compute final loss
    # -----
    # Loss for each sample:
    #   L = - (log  $\sigma(s_{\text{pos}})$  +  $\sum \log \sigma(-s_{\text{neg}})$ )
    #
    # Then average over the entire batch
    loss = - (positive_score + negative_score).mean()

    return loss

```

Excellent — this is one of the most conceptually deep parts of Word2Vec, so let's break this down **line by line**, explaining both **what the code does** and **why it exists mathematically**.

## Background: What Skip-Gram with Negative Sampling Does

Word2Vec's **Skip-Gram** model tries to learn **vector representations (embeddings)** of words such that:

Words that appear in similar contexts end up having similar vectors.

Instead of predicting *the entire vocabulary* (which is slow), **Negative Sampling** teaches the model to:

- **Increase similarity** (dot product) between a real word pair: (target, context)
- **Decrease similarity** between fake pairs: (target, random noise words)

## Code Explanation in Detail

```
import torch
import torch.nn as nn
```

- `torch` → The main PyTorch library for tensors (multi-dimensional arrays).
- `torch.nn` → Contains neural network building blocks (like layers, activations, and losses).

```
class SkipGramNegSampling(nn.Module):
```

This defines a **PyTorch neural network class** for the **Skip-Gram model using Negative Sampling**.

By inheriting from `nn.Module`, this class gains:

- trainable parameters
- automatic differentiation
- standard model behavior (`forward`, `state_dict`, etc.)

### `__init__` Constructor

```
def __init__(self, vocab_size, embedding_dim):
    super(SkipGramNegSampling, self).__init__()
```

- `vocab_size` → number of unique words in your vocabulary.
- `embedding_dim` → how many features each word vector has (e.g. 100 or 300 dimensions).
- `super()` initializes the PyTorch base class functionality.

### 1 Define model parameters

```
self.vocab_size = vocab_size
```

Stores the total number of words — useful for embedding layers.

### 2 Embedding layers

```
self.embeddings = nn.Embedding(vocab_size, embedding_dim)
```

- Creates a lookup table for **target words (center words)**. Each word ID (0 to `vocab_size-1`) maps to a learnable vector.

Example:

Word	ID	Embedding (size 3 example)
"dog"	12	[0.34, -0.17, 0.29]

```
self.context_embeddings = nn.Embedding(vocab_size, embedding_dim)
```

- Creates a *separate* lookup table for **context words** (neighbors around the target).
- Having two matrices allows the model to learn:

- `W_target` → what a word *is*
- `W_context` → how a word *is used around others*

💡 At the end, the embeddings from these two layers are often averaged or reused as the final word vectors.

### 3 Activation function

```
self.log_sigmoid = nn.LogSigmoid()
```

- Computes:  $[\log(\sigma(x)) = \log(\frac{1}{1+e^{-x}})]$
- Used because it's **numerically stable** compared to using `torch.log(torch.sigmoid(x))`.
- The skip-gram negative sampling loss formula involves:
  - `log(sigmoid(positive_score))`
  - `log(sigmoid(-negative_score))`

### ◆ forward Method — Main Logic

```
def forward(self, target, context, negative_samples):
```

- `target`: Tensor of word IDs for center words. Shape: `[batch_size]`
- `context`: Tensor of word IDs for **true context** words. Shape: `[batch_size]`
- `negative_samples`: Tensor of word IDs for **noise words**. Shape: `[batch_size, num_negatives]`

### Step 1: Embedding Lookups

```
target_embedding = self.embeddings(target)
context_embedding = self.context_embeddings(context)
negative_embeddings = self.context_embeddings(negative_samples)
```

Shapes:

Variable	Shape	Meaning
<code>target_embedding</code>	<code>[batch_size, embedding_dim]</code>	vector for each center word
<code>context_embedding</code>	<code>[batch_size, embedding_dim]</code>	vector for each true context word
<code>negative_embeddings</code>	<code>[batch_size, num_neg, embedding_dim]</code>	vectors for randomly chosen "wrong" words

### Step 2: Positive Pair Score

```
positive_score = self.log_sigmoid(
    torch.sum(target_embedding * context_embedding, dim=1)
)
```

Breakdown:

1. `target_embedding * context_embedding` → elementwise multiply vectors.
2. `torch.sum(..., dim=1)` → compute dot product for each pair (center-context).  $[s_{pos} = v_{target} \cdot v_{context}]$
3. `log_sigmoid(...)` → apply  $\log(\sigma(s_{pos}))$ , which rewards large positive dot products (i.e., similar vectors).

Intuition:

The model is learning to **increase similarity** between words that actually co-occur.

### Step 3: Negative Pair Scores

```
negative_score = self.log_sigmoid(
    -torch.bmm(negative_embeddings, target_embedding.unsqueeze(2)).squeeze(2)
).sum(1)
```

Let's decode that monster step by step 🗣️

1. `target_embedding.unsqueeze(2)` Adds a dimension → `[batch, embed_dim, 1]` Needed to perform matrix multiplication.
2. `torch.bmm(negative_embeddings, target_embedding.unsqueeze(2))` Batch matrix multiply:

- `negative_embeddings`: `[batch, num_neg, embed_dim]`
  - `target_embedding.unsqueeze(2)`: `[batch, embed_dim, 1]`
  - Output: `[batch, num_neg, 1]` → Each element = dot product between target and one negative sample.
3. `- (...)` Negates the scores (because we want `log σ(-s_neg)`).
  4. `.squeeze(2)` Removes the last dimension → `[batch, num_neg]`
  5. `self.log_sigmoid(...)` Computes `log σ(-score)` → this punishes similarity with wrong words.
  6. `.sum(1)` Sums across all negative samples per target.

Intuition:

The model **reduces similarity** between the target word and random noise words.

#### Step 4: Combine Positive + Negative Loss

```
loss = - (positive_score + negative_score).mean()
```

Mathematical formula: 
$$L = -\frac{1}{N} \sum_{i=1}^N [\log \sigma(v_{t_i} \cdot v_{c_i}) + \sum_{k=1}^K \log \sigma(-v_{t_i} \cdot v_{n_{ik}})]$$

Where:

- $(v_{t_i})$ : target word vector
- $(v_{c_i})$ : context word vector
- $(v_{n_{ik}})$ : k-th negative sample vector
- $(K)$ : number of negative samples

This loss encourages:

- Large dot product for positive pairs
- Small (negative) dot product for negative pairs

#### ◆ Concept Summary

Concept	Meaning
<b>Embedding layers</b>	Store and learn vector representations of words
<b>Positive sampling</b>	Pairs of real co-occurring words (target–context)
<b>Negative sampling</b>	Pairs of (target–random words) that <i>shouldn't</i> co-occur
<b>Dot product</b>	Measures similarity between two vectors
<b>Log-sigmoid</b>	Converts similarity into probability-like scores
<b>Loss</b>	Maximizes $\log(\sigma(\text{pos}))$ and minimizes $\log(\sigma(-\text{neg}))$

```
# -----
# TRAINING THE SKIP-GRAM MODEL WITH NEGATIVE SAMPLING
# -----

# Total number of unique words in the vocabulary
vocab_size = len(word_to_idx)

# Initialize the Skip-gram model with given vocabulary size and embedding dimension
model = SkipGramNegSampling(vocab_size, embedding_dim)

# Define the optimizer (Adam) to update the model parameters (embeddings)
# learning_rate controls how big the update steps are
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# -----
# MAIN TRAINING LOOP
# -----
for epoch in range(num_epochs):
    total_loss = 0 # To accumulate total loss for each epoch

    # Loop over all mini-batches of (target, context) pairs
    for target, context in dataloader:

        # Ensure that both tensors are of type long (required by nn.Embedding)
        target = target.long()
        context = context.long()
```

```

# -----
# NEGATIVE SAMPLING
# -----
# For each target word in the batch, randomly pick 'num_negative_samples'
# words from the vocabulary that are *not* the target.
# These are "fake" context words (noise samples).
negative_samples = torch.LongTensor([
    get_negative_samples(t.item(), num_negative_samples, vocab_size)
    for t in target
])

# -----
# FORWARD AND BACKWARD PASSES
# -----

# 1. Clear any previously stored gradients before backpropagation
optimizer.zero_grad()

# 2. Compute the model loss for this batch
# The model internally:
#   - Looks up embeddings for targets, contexts, and negatives
#   - Computes positive and negative log-sigmoid scores
#   - Returns average loss for the batch
loss = model(target, context, negative_samples)

# 3. Compute gradients ( $\partial \text{Loss} / \partial \text{Parameters}$ )
# This step performs automatic differentiation
loss.backward()

# 4. Update embeddings using gradients
optimizer.step()

# -----
# TRACKING PROGRESS
# -----

# Add current batch loss to the total loss for this epoch
total_loss += loss.item()

# -----
# DISPLAY AVERAGE EPOCH LOSS
# -----
print(f"Epoch {epoch + 1}, Loss: {total_loss / len(dataloader):.4f}")

```

```

Epoch 1, Loss: 23.6487
Epoch 2, Loss: 23.7863
Epoch 3, Loss: 23.3079
Epoch 4, Loss: 22.4350
Epoch 5, Loss: 23.8155

```

```

# -----
# GETTING THE LEARNED WORD EMBEDDINGS
# -----

# Extract the learned embeddings from the model
# model.embeddings.weight → tensor containing learned word vectors for all words
# detach() → disconnects from computation graph (no gradients needed now)
# numpy() → converts PyTorch tensor into a NumPy array for easier math operations
embeddings = model.embeddings.weight.detach().numpy()

# -----
# FUNCTION: FIND SIMILAR WORDS USING COSINE SIMILARITY
# -----

def get_similar_words(word, top_n=5):
    """
    Given a word, find the top-N most similar words based on cosine similarity.
    """

    # -----
    # Step 1: Get the embedding vector for the given word
    # -----
    idx = word_to_idx[word]          # Get the word's numeric ID
    word_embedding = embeddings[idx]  # Retrieve its vector representation

    # -----

```

```

# Step 2: Compute similarity scores with all other words
# -----
# np.dot(embeddings, word_embedding) computes the dot product
# between the target word and every other word in the vocabulary.
# This is equivalent to:
#   similarity(i) = v_i · v_target
#
# Note: For true cosine similarity, you'd usually normalize vectors,
# but since Word2Vec vectors tend to be roughly normalized already,
# this approximation often works fine.
similarities = np.dot(embeddings, word_embedding)

# -----
# Step 3: Sort words by similarity (highest first)
# -----
# argsort() returns indices sorted in ascending order.
# By adding a negative sign (-similarities), we sort in descending order.
# [1:top_n+1] skips the first element, which is the word itself.
closest_idxs = (-similarities).argsort()[1:top_n + 1]

# -----
# Step 4: Convert indices back to readable words
# -----
# idx_to_word maps numeric indices back to word strings
return [idx_to_word[idx] for idx in closest_idxs]

# -----
# EXAMPLE USAGE
# -----

# Print top 5 words most similar to "do"
print(get_similar_words("do"))

```

```
['excellence', 'think', 'strike', 'repeatedly', 'can']
```

Start coding or [generate](#) with AI.

Start coding or [generate](#) with AI.