What is Continuous Bag of Words Model?

The Continuous Bag of Words (CBOW) is also a model that is used when determining word embedding using a neural network and is part of Word2Vec models by Tomas Mikolov. CBOW tries to predict a target word depending on the context words observing it in a given sentence. This way it is able to capture the semantic relations hence close words are represented closely in a high dimensional space.

For example, in the sentence **“The cat sat on the mat”**, if the context window size is 2, the context words for **“sat”** are **[“The”, “cat”, “on”, “the”]**, and the model’s task is to predict the word **“sat”**.

CBOW operates by aggregating the context words (e.g., averaging their embeddings) and using this aggregate representation to predict the target word. The model’s architecture involves an input layer for the context words, a hidden layer for embedding generation, and an output layer to predict the target word using a probability distribution.

It is a fast and efficient model suitable for handling frequent words, making it ideal for tasks requiring semantic understanding, such as text classification, recommendation systems, and sentiment analysis.

How Continuous Bag of Words Works

CBOW is one of the simplest, yet efficient techniques as per context for word embedding where the whole vocabulary of words are mapped to vectors. This

section also describes the operation of the CBOW system as a means of comprehending the method at its most basic level, discussing the main ideas that underpin the CBOW method, as well as offering a comprehensive guide to the architectural layout of the CBOW hit calculation system.

Understanding Context and Target Words

CBOW relies on two key concepts: context words and the target word.

- **Context Words**: These are the words surrounding a target word within a defined window size. For example, in the sentence: **“The quick brown fox jumps over the lazy dog”**, if the target word is **“fox”** and the context window size is 2, the context words are **[“quick”, “brown”, “jumps”, “over”]**.
- **Target Word**: This is the word that CBOW aims to predict, given the context words. In the above example, the target word is **“fox”**.

By analyzing the relationship between context and target words across large corpora, CBOW generates embeddings that capture semantic relationships between words.

Step-by-Step Process of CBOW

Here’s a breakdown of how CBOW works, step-by-step:

Step1: Data Preparation

- Choose a corpus of text (e.g., sentences or paragraphs).

- Tokenize the text into words and build a vocabulary.
- Define a context window size nnn (e.g., 2 words on each side).

Step2: Generate Context-Target Pairs

- For each word in the corpus, extract its surrounding context words based on the window size.
- Example: For the sentence **“I love machine learning”** and n=2n = 2n=2, the pairs are:**Target WordContext Words**love[“I”, “machine”]machine[“love”, “learning”]

Step3: One-Hot Encoding

Convert the context words and target word into one-hot vectors based on the vocabulary size. For a vocabulary of size 5, the one-hot representation of the word “love” might look like **[0, 1, 0, 0, 0]**.

Step4: Embedding Layer

Pass the one-hot encoded context words through an embedding layer. This layer maps each word to a dense vector representation, typically of a lower dimension than the vocabulary size.

Step5: Context Aggregation

Aggregate the embeddings of all context words (e.g., by averaging or summing them) to form a single context vector.

Step6: Prediction

- Feed the aggregated context vector into a fully connected neural network with a softmax output layer.
- The model predicts the most probable word as the target based on the probability distribution over the vocabulary.

Step7: Loss Calculation and Optimization

- Compute the error between the predicted and actual target word using a cross-entropy loss function.
- Backpropagate the error to adjust the weights in the embedding and prediction layers.

Step8: Repeat for All Pairs

Repeat the process for all context-target pairs in the corpus until the model converges.

CBOW Architecture Explained in Detail

The Continuous Bag of Words (CBOW) model's architecture is designed to predict a target word based on its surrounding context words. It is a shallow neural network with a straightforward yet effective structure. The CBOW architecture consists of the following components:

Input Layer

- **Input Representation**:

  The input to the model is the context words represented as **one-hot encoded vectors**.

  - If the vocabulary size is V, each word is represented as a one-hot vector of size V with a single 1 at the index corresponding to the word, and 0s elsewhere.
  - For example, if the vocabulary is [“cat”, “dog”, “fox”, “tree”, “bird”] and the word “fox” is the third word, its one-hot vector is [0,0,1,0,0][0, 0, 1, 0, 0][0,0,1,0,0].

- **Context Window**:

  The context window size n determines the number of context words used. If n=2, two words on each side of the target word are used.

  - For a sentence: **“The quick brown fox jumps over the lazy dog”** and target word **“fox”**, the context words with n=2 are **[“quick”, “brown”, “jumps”, “over”]**.

Embedding Layer

- **Purpose**:

  This layer converts one-hot vectors which exist in a high dimension into maximally dense and low dimensions vectors. In contrast to the fact that in word embedding words are represented as vectors with mostly zero values, in the embedding layer, each word is encoded by the continuous vector of the required dimensions that reflects specific characteristics of the word meaning.

- **Word Embedding Matrix**:

  The embedding layer maintains a word embedding matrix W of size V×d, where V is the vocabulary size and d is the embedding dimension.

  - Each row of W represents the embedding of a word.
  - For a one-hot vector xxx, the embedding is computed as W^T X x.

- **Context Word Embeddings**:

  Each context word is transformed into its corresponding dense vector using the embedding matrix. If the window size n=2, and we have 4 context words, the embeddings for these words are extracted.

Hidden Layer: Context Aggregation

- **Purpose**:

  The embeddings of the context words are combined to form a single context vector.

- **Aggregation Methods**:

  - **Averaging**: The embeddings of all context words are averaged to compute the context vector.

- **Summation**: Instead of averaging, the embeddings are summed.

- **Resulting Context Vector**: The result is a single dense vector hhh, which represents the aggregated context of the surrounding words.

Output Layer

- **Purpose**: The output layer predicts the target word using the context vector hhh.
- **Fully Connected Layer**: The context vector hhh is passed through a fully connected layer, which outputs a raw score for each word in the vocabulary. These scores are called logits.
- **Softmax Function**: The logits are passed through a softmax function to compute a probability distribution over the vocabulary:

- **Predicted Target Word**: The first cause is that at the softmax output, the algorithm defines the target word as the word with the highest probability.

Loss Function

- The cross-entropy loss is used to compare the predicted probability distribution with the actual target word (ground truth).

- The loss is minimized using optimization techniques like Stochastic Gradient Descent (SGD) or its variants.

Example of CBOW in Action

Input:

Sentence: **“I love machine learning”**, target word: **“machine”**, context words: **[“I”, “love”, “learning”]**.

One-Hot Encoding:

Vocabulary: **[“I”, “love”, “machine”, “learning”, “AI”]**

- One-hot vectors:
  - “I”: [1,0,0,0,0][1, 0, 0, 0, 0][1,0,0,0,0]
  - “love”: [0,1,0,0,0][0, 1, 0, 0, 0][0,1,0,0,0]
  - “learning”: [0,0,0,1,0][0, 0, 0, 1, 0][0,0,0,1,0]

Embedding Layer:

- Embedding dimension: d=3.
- Embedding matrix W:

Embeddings:

- “I”: [0.1,0.2,0.3]
- “love”: [0.4,0.5,0.6]
- “learning”: [0.2,0.3,0.4]

Aggregation:

- Average the embeddings:

Output Layer:

- Compute logits, apply softmax, and predict the target word.

Diagram of CBOW Architecture

```
Input Layer: ["I", "love", "learning"]
    --> One-hot encoding
    --> Embedding Layer
        --> Dense embeddings
        --> Aggregated context vector
        --> Fully connected layer + Softmax
Output: Predicted word "machine"
```

Coding CBOW from Scratch (with Python Examples)

We'll now walk through implementing the CBOW model from scratch in Python.

Preparing Data for CBOW

The first spike is to transform the text into tokens, words that are generated into context-target pairs with context as the words containing the target word.

```
corpus = "The quick brown fox jumps over the lazy dog"
corpus = corpus.lower().split()  # Tokenization and lowercase conversion

# Define context window size
C = 2
context_target_pairs = []

# Generate context-target pairs
for i in range(C, len(corpus) - C):
    context = corpus[i - C:i] + corpus[i + 1:i + C + 1]
    target = corpus[i]
    context_target_pairs.append((context, target))

print("Context-Target Pairs:", context_target_pairs)
```

Output:

```
Context-Target Pairs: [(['the', 'quick', 'fox', 'jumps'], 'brown'), (['quick', 'brown', 'jumps', 'over'], 'fox'), (['brown', 'fox', 'over', 'the'], 'jumps'), (['fox', 'jumps', 'the', 'lazy'], 'over'), (['jumps', 'over', 'lazy', 'dog'], 'the')]
```

Creating the Word Dictionary

We build a vocabulary (a unique set of words), then map each word to a unique index and vice versa for efficient lookups during training.

```
# Create vocabulary and map each word to an index
vocab = set(corpus)
word_to_index = {word: idx for idx, word in enumerate(vocab)}
index_to_word = {idx: word for word, idx in word_to_index.items()}

print("Word to Index Dictionary:", word_to_index)
```

Output:

```
Word to Index Dictionary: {'brown': 0, 'dog': 1, 'quick': 2, 'jumps': 3, 'fox': 4, 'over': 5, 'the': 6, 'lazy': 7}
```

One-Hot Encoding Example

One-hot encoding works by transforming each word in the word formation system into a vector, where the indicator of the word is ‘1’ while the rest of the places take ‘0,’ for reasons that shall soon be clear.

```
def one_hot_encode(word, word_to_index):
    one_hot = np.zeros(len(word_to_index))
    one_hot[word_to_index[word]] = 1
    return one_hot

# Example usage for a word "quick"
context_one_hot = [one_hot_encode(word, word_to_index) for word in ['the', 'quick']]
print("One-Hot Encoding for 'quick':", context_one_hot[1])
```

Output:

```
One-Hot Encoding for 'quick': [0. 0. 1. 0. 0. 0. 0. 0.]
```

Building the CBOW Model from Scratch

In this step, we create a basic neural network with two layers: one for word embeddings and another to compute the output based on context words, averaging the context and passing it through the network.

```
class CBOW:
    def __init__(self, vocab_size, embedding_dim):
        # Randomly initialize weights for the embedding and output layers
        self.W1 = np.random.randn(vocab_size, embedding_dim)
        self.W2 = np.random.randn(embedding_dim, vocab_size)

    def forward(self, context_words):
        # Calculate the hidden layer (average of context words)
        h = np.mean(context_words, axis=0)
        # Calculate the output layer (softmax probabilities)
        output = np.dot(h, self.W2)
```

```
        return output

    def backward(self, context_words, target_word, learning_rate=0.01):
        # Forward pass
        h = np.mean(context_words, axis=0)
        output = np.dot(h, self.W2)

        # Calculate error and gradients
        error = target_word - output
        self.W2 += learning_rate * np.outer(h, error)
        self.W1 += learning_rate * np.outer(context_words, error)

# Example of creating a CBOW object
vocab_size = len(word_to_index)
embedding_dim = 5  # Let's assume 5-dimensional embeddings

cbow_model = CBOW(vocab_size, embedding_dim)

# Using random context words and target (as an example)
context_words = [one_hot_encode(word, word_to_index) for word in ['the', 'quick',
'fox', 'jumps']]
context_words = np.array(context_words)
context_words = np.mean(context_words, axis=0)  # average context words
target_word = one_hot_encode('brown', word_to_index)

# Forward pass through the CBOW model
output = cbow_model.forward(context_words)
print("Output of CBOW forward pass:", output)
```

Output:

```
Output of CBOW forward pass: [[-0.20435729 -0.23851241 -0.08105261 -0.14251447
0.20442154  0.14336586
  -0.06523201  0.0255063 ]
 [-0.0192184  -0.12958821  0.1019369   0.11101922 -0.17773069 -0.02340574
  -0.22222151 -0.23863179]
 [ 0.21221977 -0.15263454 -0.015248    0.27618767  0.02959409  0.21777961
   0.16619577 -0.20560026]
 [ 0.05354038  0.06903295  0.0592706  -0.13509918 -0.00439649  0.18007843
   0.1611929   0.2449023 ]
 [ 0.01092826  0.19643582 -0.07430934 -0.16443165 -0.01094085 -0.27452367
  -0.13747784  0.31185284]]
```

Using TensorFlow to Implement CBOW

TensorFlow simplifies the process by defining a neural network that uses an embedding layer to learn word representations and a dense layer for output, using context words to predict a target word.

```
import tensorflow as tf

# Define a simple CBOW model using TensorFlow
class CBOWModel(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim):
        super(CBOWModel, self).__init__()
        self.embeddings     =     tf.keras.layers.Embedding(input_dim=vocab_size,
output_dim=embedding_dim)
        self.output_layer           =           tf.keras.layers.Dense(vocab_size,
activation='softmax')

    def call(self, context_words):
        embedded_context = self.embeddings(context_words)
        context_avg = tf.reduce_mean(embedded_context, axis=1)
        output = self.output_layer(context_avg)
        return output

# Example usage
model = CBOWModel(vocab_size=8, embedding_dim=5)
context_input = np.random.randint(0, 8, size=(1, 4))  # Random context input
context_input = tf.convert_to_tensor(context_input, dtype=tf.int32)

# Forward pass
output = model(context_input)
print("Output of TensorFlow CBOW model:", output.numpy())
```

Output:

```
Output of TensorFlow CBOW model: [[0.12362909 0.12616573 0.12758036 0.12601459
0.12477358 0.1237749
  0.12319998 0.12486169]]
```

Using Gensim for CBOW

Gensim offers ready-made implementation of CBOW in the Word2Vec() function where one does not need to labor on training as Gensim trains word embeddings from a corpus of text.

```
import gensim
from gensim.models import Word2Vec

# Prepare data (list of lists of words)
corpus = [["the", "quick", "brown", "fox"], ["jumps", "over", "the", "lazy",
"dog"]]

# Train the Word2Vec model using CBOW
model = Word2Vec(corpus, vector_size=5, window=2, min_count=1, sg=0)
```

```
# Get the vector representation of a word
vector = model.wv['fox']
print("Vector representation of 'fox':", vector)
```

Output:

```
Vector representation of 'fox': [-0.06810732 -0.01892803  0.11537147 -0.15043275
-0.07872207]
```

Advantages of Continuous Bag of Words

We will now explore advantages of continuous bag of words:

- **Efficient Learning of Word Representations**: CBOW efficiently learns dense vector representations for words by using context words. This results in lower-dimensional vectors compared to traditional one-hot encoding, which can be computationally expensive.
- **Captures Semantic Relationships**: CBOW captures semantic relationships between words based on their context in a large corpus. This allows the model to learn word similarities, synonyms, and other contextual nuances, which are useful in tasks like information retrieval and sentiment analysis.
- **Scalability**: The CBOW model is highly scalable and can process large datasets efficiently, making it well-suited for applications with vast amounts of text data, such as search engines and social media platforms.
- **Contextual Flexibility**: CBOW can handle varying amounts of context (i.e., the number of surrounding words considered), offering flexibility in how much context is required for learning the word representations.

- **Improved Performance in NLP Tasks**: CBOW's word embeddings enhance the performance of downstream NLP tasks, such as text classification, named entity recognition, and machine translation, by providing high-quality feature representations.

Limitations of Continuous Bag of Words

Let us now discuss the limitations of CBOW:

- **Sensitivity to Context Window Size**: The performance of CBOW is highly dependent on the context window size. A small window may result in capturing only local relationships, while a large window may blur the distinctiveness of words. Finding the optimal context size can be challenging and task-dependent.
- **Lack of Word Order Sensitivity**: CBOW disregards the order of words within the context, meaning it doesn't capture the sequential nature of language. This limitation can be problematic for tasks that require a deep understanding of word order, like **syntactic parsing** and **language modeling**.
- **Difficulty with Rare Words**: CBOW struggles to generate meaningful embeddings for rare or out-of-vocabulary (OOV) words. The model relies on context, but sparse data for infrequent words can lead to poor vector representations.
- **Limited to Shallow Contextual Understanding**: While CBOW captures word meanings based on surrounding words, it has limited capabilities in

understanding more complex linguistic phenomena, such as long-range dependencies, irony, or sarcasm, which may require more sophisticated models like transformers.

- **Inability to Handle Polysemy Well**: Words with multiple meanings (polysemy) can be problematic for CBOW. Since the model generates a single embedding for each word, it may not capture the different meanings a word can have in different contexts, unlike more advanced models like **BERT** or **ELMo**.

Conclusion

The Continuous Bag of Words (CBOW) model has proven to be an efficient and intuitive approach for generating word embeddings by leveraging surrounding context. Through its simple yet effective architecture, CBOW bridges the gap between raw text and meaningful vector representations, enabling a wide range of NLP applications. By understanding CBOW's working mechanism, its strengths, and limitations, we gain deeper insights into the evolution of NLP techniques. With its foundational role in embedding generation, CBOW continues to be a stepping stone for exploring advanced language models.

Key Takeaways

- CBOW predicts a target word using its surrounding context, making it efficient and simple.
- It works well for frequent words, offering computational efficiency.

- The embeddings learned by CBOW capture both semantic and syntactic relationships.
- CBOW is foundational for understanding modern word embedding techniques.
- Practical applications include sentiment analysis, semantic search, and text recommendations.

==========

Google’s Word2vec Pretrained Word Embedding

Word2Vec is one of the most popular pretrained word embeddings developed by Google. Word2Vec is trained on the Google News dataset (about 100 billion words). It has several use cases, such as Recommendation Engines, and Knowledge Discovery, and is also applied to different Text Classification problems.

Word2Vec’s architecture is really simple. It’s a feed-forward neural network with just one hidden layer, sometimes called a **Shallow Neural Network architecture**.

Depending on the way the embeddings are learned, Word2Vec is classified into two approaches:

- Continuous Bag-of-Words (CBOW)
- Skip-gram model

The continuous Bag-of-Words (CBOW) model learns the focus word given the neighbouring words, whereas the Skip-gram model learns the neighbouring words given the focus word. That's why:

Continuous Bag Of Words and Skip-gram are inverses of each other

Example

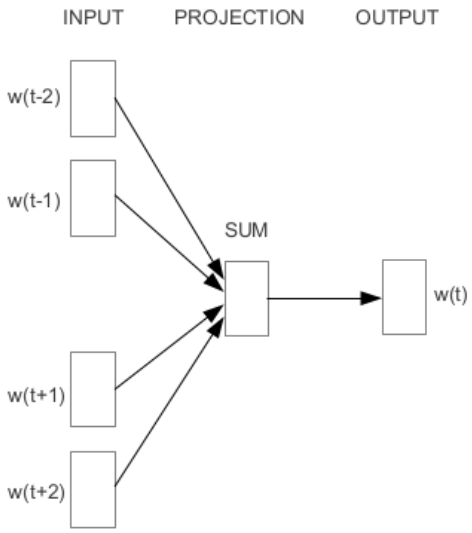
For example, consider the sentence: "I have failed at times, but I never stopped trying". Let's say we want to learn the embedding of the word "failed". So, here, the focus word is "failed".

The first step is to define a context window. A context window refers to the number of words appearing on the left and right of a focus word. The words appearing in the context window are known as neighbouring words (or context). Let's fix the context window to 2 and then input and output pairs for both approaches:

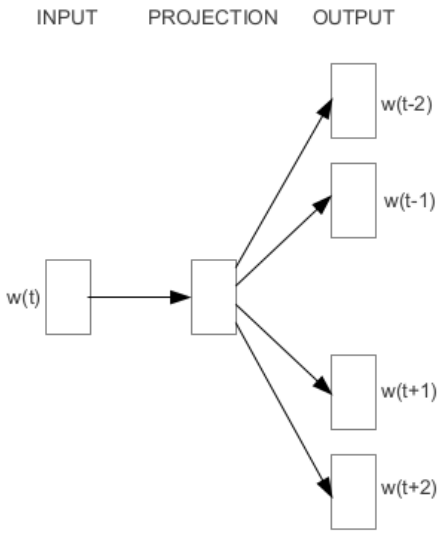
- **Continuous Bag-of-Words:** Input = [I, have, at, times], Output = failed
- **Skip-gram:** Input = failed, Output = [I, have, at, times]

As you can see here, CBOW accepts multiple words as input and produces a single word as output, whereas Skip-gram accepts a single word as input and produces multiple words as output.

So, let us define the architecture according to the input and output. But keep in mind that each word is fed into a model as a one-hot vector:



CBOW



Skip-gram

Skip- gram model

In a nutshell, this algorithm maps words to vector representation based on their spatial proximity to other words. It is important to note that the underlying assumption here is that if words are synthetically positioned next to each other, it means that they are also semantically similar. Although it's not always the case — as you would undoubtedly agree — this algorithm produces good outcomes.

The way this algorithm works is by iterating over all the words in our text input and optimizing their vector representations distance based on their context. The context is defined as a window of words surrounding a specific word. For instance, addressing the sentence:

“ I don’t do magic, **Morty**, I do science”

If the word ‘Morty’ is our target (center) word, and our context window size is set to 1, the neighboring words would be ‘magic’ and ‘I’. All these pairs of (target word, context word) can be then fed into the model and by that optimizing the word embeddings. This indicates that each pair of words adds the same amount to the model.

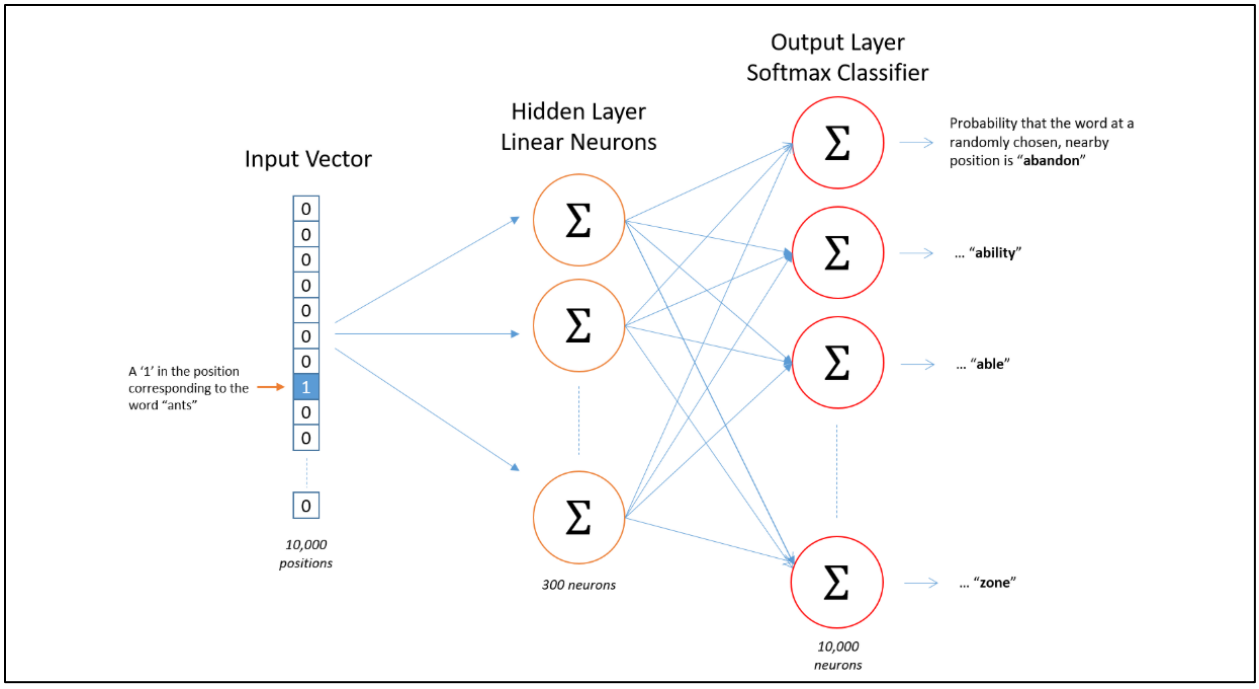
How Skip-Gram works?

(1) Model Structure:

The Skip-Gram model is basically a shallow neural network with an input layer, embedding layer and output layer. The model’s aim is to produce an output probability distribution vector given a target word input. This probability distribution vector (which sums to 1) reflects the probability for each word to appear in the target word’s context window. As one might expect, we are interested in locating the probability to be high for words who share the same context and low for words who don’t.

The intriguing part about this is that we just require the model’s weights once it has been trained.

Architecture of skip-gram model network.



1. Input Vector (One-Hot Encoding)

- Each word in the vocabulary (say 10,000 words) is represented by a **one-hot vector**.
- A one-hot vector has all zeros except for a **1** at the index corresponding to the current (input) word.
- Example here:
  The word **"ants"** is represented by a vector of length 10,000 with a 1 at the position for "ants".

2. Hidden Layer (Linear Neurons)

- This layer has **300 neurons** (this number is the embedding dimension).
- There are **no activation functions** here — just a **linear transformation**:

$$h = W_1^T \times x$$

where

- x = input one-hot vector (10,000 × 1)
- W_1 = weight matrix (10,000 × 300)
- h = hidden layer output (300 × 1), which becomes the **word embedding** for "ants".

➤ So this layer learns a **dense vector representation** (embedding) for each word.

3. Output Layer (Softmax Classifier)

- The hidden layer output is then multiplied by another weight matrix W2 to produce a **score for each word** in the vocabulary:

$$u = W_2^T \times h$$

where u is a 10,000-dimensional vector (one score per possible output word).

4. Softmax Function

This is where the **Softmax** function comes in.

It converts the raw scores ui into **probabilities** that sum to 1:

⚙️ 4. Softmax Function

This is where the **Softmax** function comes in.

It converts the raw scores u_i into **probabilities** that sum to 1:

$$P(w_i|w_{\text{input}}) = \frac{e^{u_i}}{\sum_{j=1}^{10,000} e^{u_j}}$$

- $P(w_i|w_{\text{input}})$: probability that word w_i appears near the input word.
- The higher the u_i, the higher the probability.

🔍 5. Output Interpretation

- After applying softmax, you get a **probability distribution** over all words in the vocabulary.

- For example:
  - P("abandon") = 0.35
  - P("ability") = 0.25
  - P("able") = 0.20
  - P("zone") = 0.05
  - … (all add up to 1)

The model tries to predict which **nearby word** (context word) is most likely to appear given the input word.

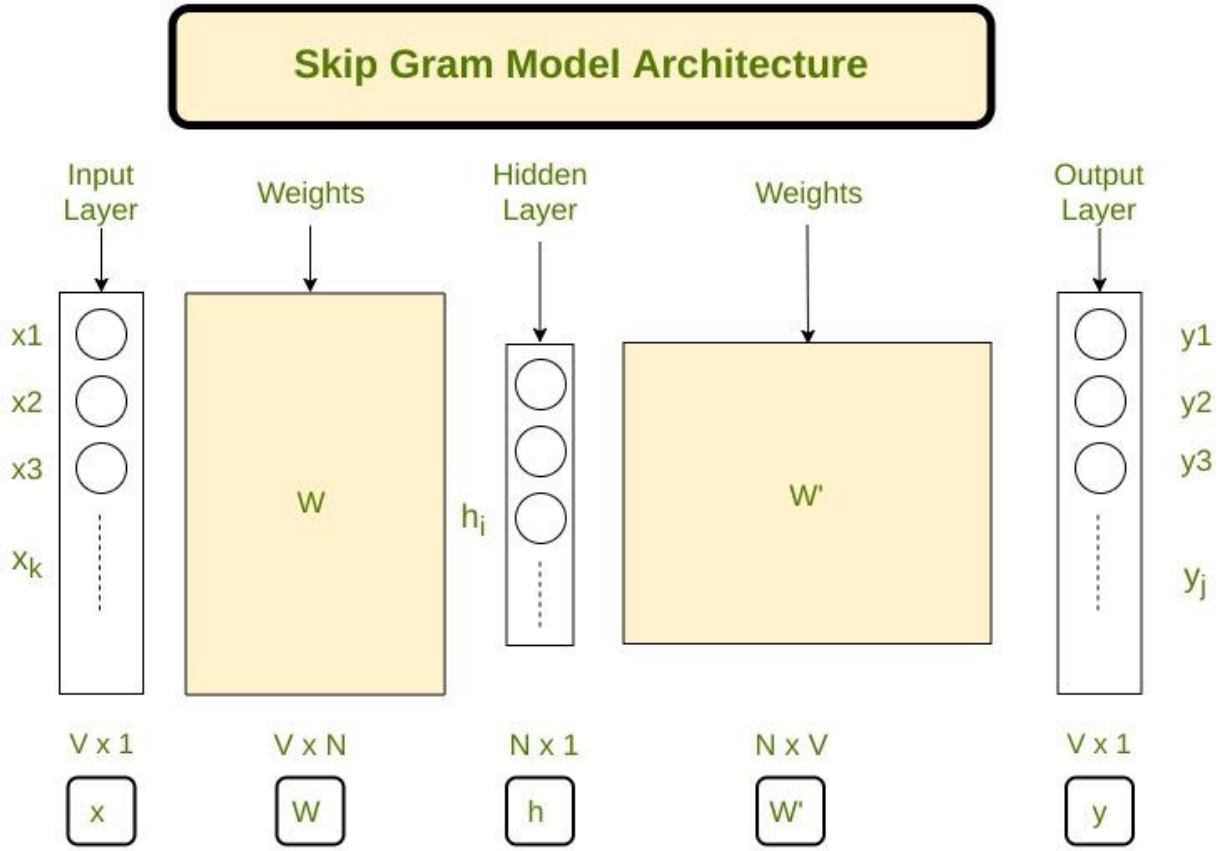
Purpose of the Softmax Function

The **softmax** layer:

- Converts arbitrary scores into probabilities (range 0–1).
- Ensures all probabilities sum to 1.
- Helps the model learn by comparing predicted probabilities to the true context word (using cross-entropy loss).

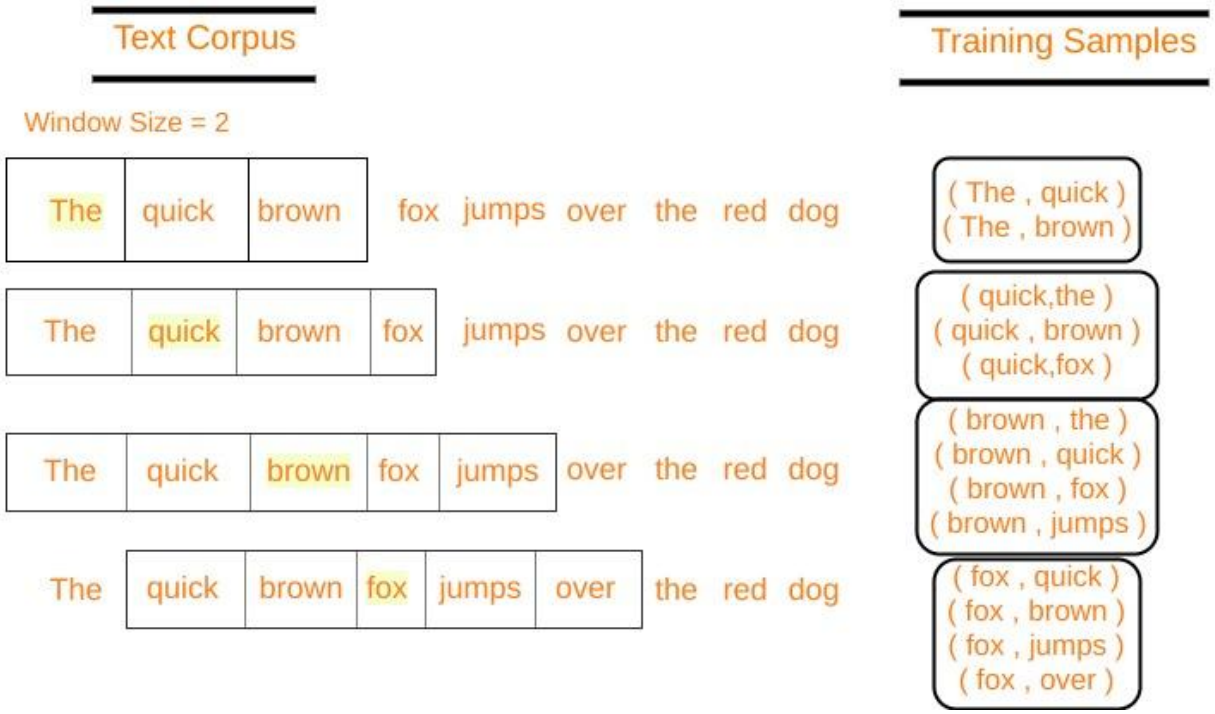
Summary

Layer	Purpose	Output
Input Layer	One-hot encode the target word	Vector of size 10,000
Hidden Layer	Learn word embeddings (dense representation)	300-dimensional embedding
Output Layer (Softmax)	Predict probability distribution over all words	Vector of size 10,000 (probabilities)



What is Skip-Gram?

In **skip-gram** architecture of word2vec, the input is the **center word** and the predictions are the context words. Consider an array of words W, if W(i) is the input (center word), then W(i-2), W(i-1), W(i+1), and W(i+2) are the context words if the *sliding window size* is 2.



Let's define some variables :

V Number of unique words in our corpus of text (**V**ocabulary)
x Input layer (One hot encoding of our input word).
N Number of neurons in the hidden layer of neural network
W Weights between input layer and hidden layer
W' Weights between hidden layer and output layer
y A softmax output layer having probabilities of every word in our vocabulary

Forward Propagation

1. **Hidden Layer Calculation**:
   - Multiply one hot encoding of the center word (denoted by x) with the first weight matrix W to get hidden layer matrix h.
   - $h = W^T.x$
   - Dimensions: $(V \times 1)(N \times V)(V \times 1)$

2. **Output Layer Calculation**:
   - Multiply the hidden layer vector h with second weight matrix W' to get a new matrix u.
   - $u = W'^T.h$
   - Dimensions: $(V \times 1)(V \times N)(N \times 1)$

3. **Apply Softmax**:

- Apply a softmax function to layer u to get our output layer y.
- Let u_j be the j[th] neuron of layer u.
- Let w_j be the j[th] word in our vocabulary where j is any index.
- Let V_{w_j} be the jth column of matrix W' (column corresponding to a word w_j).
- $u_j = V_{w_j}^T . h$
- $y = \text{softmax}(u)$
- $y_j = \text{softmax}(u_j)$
- $P(w_j|w_i) = y_j = \frac{e^{u_j}}{\sum_{j'=1}^{V} e^{u_{j'}}}$

4. **Maximizing Probability**:

- Our goal is to maximize $P(w_{j*}|w_i)$, where j^* represents the indices of context words.
- Maximize $\prod_{c=1}^{C} \frac{e^{u_{j_c^*}}}{\sum_{j'=1}^{V} e^{u_{j'}}}$

Loss Function

- Take the negative log-likelihood of this function to get our loss function.
- $E = -\log\left(\prod_{c=1}^{C} \frac{e^{u_{j_c^*}}}{\sum_{j'=1}^{V} e^{u_{j'}}}\right)$
- Let t be the actual output vector from our training data.
- $E = -\sum_{c=1}^{C} u_{j_c^*} + C \log \sum_{j'=1}^{V} e^{u_{j'}}$

Back Propagation

The parameters to be adjusted are in the matrices W and W', hence we have to find the partial derivatives of our loss function with respect to W and W' to apply the gradient descent algorithm.

Derivatives Calculation

1. **For W'**:
   - $\frac{\partial E}{\partial W'} = \frac{\partial E}{\partial u_j} \cdot \frac{\partial u_j}{\partial W'}$
   - $\frac{\partial E}{\partial u_j} = y_j - t_j = e_j$
   - $\frac{\partial E}{\partial W'} = e_j \cdot h_i$

2. **For W**:
   - $\frac{\partial E}{\partial W} = \frac{\partial E}{\partial u_j} \cdot \frac{\partial u_j}{\partial h_i} \cdot \frac{\partial h_i}{\partial W}$
   - $\frac{\partial E}{\partial W} = e_j \cdot W' \cdot x_i$